Android Developer Fundamentals Course

Learn to develop Android Applications

Concept Reference

Developed by Google Developer Training Team

December 2016

Table of Contents

- Introduction 1.1
- Unit 1. Get started 1.2
  - Lesson 1: Build your first app 1.2.1
    - 1.0: Introduction to Android 1.2.1.1
    - 1.1: Create Your First Android App 1.2.1.2
    - 1.2: Layouts, Views and Resources 1.2.1.3
    - 1.3: Text and Scrolling Views 1.2.1.4
    - 1.4: Resources to Help You Learn 1.2.1.5
  - Lesson 2: Activities 1.2.2
    - 2.1: Understanding Activities and Intents 1.2.2.1
    - 2.2: The Activity Lifecycle and Managing State 1.2.2.2
    - 2.3: Activities and Implicit Intents 1.2.2.3
  - Lesson 3: Testing, debugging, and using support libraries 1.2.3
    - 3.1: The Android Studio Debugger 1.2.3.1
    - 3.2: Testing your App 1.2.3.2
    - 3.3: The Android Support Library 1.2.3.3
- Unit 2. User experience 1.3
  - Lesson 4: User interaction 1.3.1
    - 4.1: User Input Controls 1.3.1.1
    - 4.2: Menus 1.3.1.2
    - 4.3: Screen Navigation 1.3.1.3
    - 4.4: RecyclerView 1.3.1.4
  - Lesson 5: Delightful user experience 1.3.2
    - 5.1: Drawables, Styles, and Themes 1.3.2.1
    - 5.2: Material Design 1.3.2.2
    - 5.3: Providing Resources for Adaptive Layouts 1.3.2.3
  - Lesson 6: Testing your UI 1.3.3
    - 6.1: Testing the User Interface 1.3.3.1
- Unit 3. Working in the background 1.4
  - Lesson 7: Background Tasks 1.4.1
    - 7.1: AsyncTask and AsyncTaskLoader 1.4.1.1
    - 7.2: Connect to the Internet 1.4.1.2
    - 7.3: Broadcast Receivers 1.4.1.3
    - 7.4: Services 1.4.1.4
  - Lesson 8: Triggering, scheduling and optimizing background tasks 1.4.2
    - 8.1: Notifications 1.4.2.1
    - 8.2: Scheduling Alarms 1.4.2.2
    - 8.3: Transferring Data Efficiently 1.4.2.3
- Unit 4. All about data 1.5

- Lesson 9: Preferences and Settings 1.5.1
  - 9.0: Storing Data 1.5.1.1
  - 9.1: Shared Preferences 1.5.1.2
  - 9.2: App Settings 1.5.1.3
- Lesson 10: Storing data using SQLite 1.5.2
  - 10.0: SQLite Primer 1.5.2.1
  - 10.1: SQLite Database 1.5.2.2
- Lesson 11: Sharing data with content providers 1.5.3
  - 11.1: Share Data Through Content Providers 1.5.3.1
- Lesson 12: Loading data using loaders 1.5.4
  - 12.1: Loaders 1.5.4.1

Unit 5. What's Next? 1.6

- Lesson 13: Permissions, Performance and Security 1.6.1
  - 13.1: Permissions, Performance and Security 1.6.1.1
- Lesson 14: Firebase and AdMob 1.6.2
  - 14.1: Firebase and AdMob 1.6.2.1
- Lesson 15: Publish! 1.6.3
  - 15.1: Publish! 1.6.3.1

Android Developer Fundamentals Course – Concepts

Android Developer Fundamentals is a training course created by the Google Developer Training team. You learn basic Android programming concepts and build a variety of apps, starting with Hello World and working your way up to apps that use content providers and loaders.

Android Developer Fundamentals prepares you to take the exam for the Associate Android Developer Certification.

This course is intended to be taught in a classroom, but all the materials are online, so if you like to learn by yourself, go ahead!

Prerequisites

Android Developer Fundamentals is intended for new and experienced developers who already have Java programming experience and now want to learn to build Android apps.

Course materials

The course materials include:

- This concept reference, which teaches subjects you need to learn to complete the exercises in the practical workbook. Some lessons are purely conceptual and do not have an accompanying practical.
- The practical workbook: Android Developer Fundamentals Course—Practicals
- Slide decks (for optional use by instructors)
- Videos of lectures (for reference by instructors and students)

What topics are covered?

Android Developer Fundamentals includes five teaching units, which are described in What does the course cover?

Developed by the Google Developer Training Team

Last updated: February 2017

This work is licensed under a Creative Commons Attribution-Non Commercial 4.0 International License

1.0: Introduction to Android

Contents:

- What is Android?
- Why develop apps for Android?
- Android versions
- The challenges of Android app development
- Learn more

What is Android?

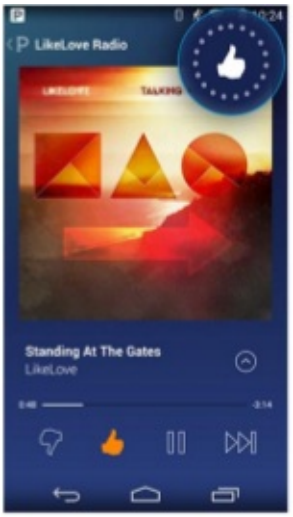
Android is an operating system and programming platform developed by Google for smartphones and other mobile devices (such as tablets). It can run on many different devices from many different manufacturers. Android includes a software development kit for writing original code and assembling software modules to create apps for Android users. It also provides a marketplace to distribute apps. All together, Android represents an ecosystem for mobile apps.

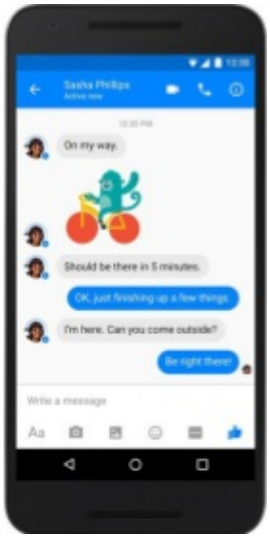
Why develop apps for Android?

Apps are developed for a variety of reasons: addressing business requirements, building new services, creating new businesses, and providing games and other types of content for users. Developers choose to develop for Android in order to reach the majority of mobile device users.

Most popular platform for mobile apps

As the world's most popular mobile platform, Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It has the largest installed base of any mobile platform and is still growing fast. Every day another million users power up their Android devices for the first time and start looking for apps, games, and other digital




On my way.
Should be there in 5 minutes.
OK, just finishing up a few things.
I'm here. Can you come outside?
Be right there!
Write a message

content.

Best experience for app users

Android provides a touch-screen user interface (UI) for interacting with apps. Android's user interface is mainly based on direct manipulation, using touch gestures such as swiping, tapping and pinching to manipulate on-screen objects. In addition to the keyboard, there's a customizable virtual keyboard for text input. Android can also support game controllers and full-size physical keyboards connected by Bluetooth or USB.

The Android home screen can contain several pages of *app icons*, which launch the associated apps, and *widgets*, which display live, auto-updating content such as the weather, the user's email inbox or a news ticker. Android can also play multimedia content such as music, animation, and video. The figure above shows app icons on the home screen (left), playing music (center), and displaying widgets (right). Along the top of the screen is a status bar, showing information about the device and its connectivity. The Android home screen may be made up of several pages, between which the user can swipe back and forth.

Android is designed to provide immediate response to user input. Besides a fluid touch interface, the vibration capabilities of an Android device can provide haptic feedback. Internal hardware such as accelerometers, gyroscopes and proximity sensors, are used by many apps to respond to additional user actions. These sensors can detect rotation of the screen from portrait to landscape for a wider view or it can allow the user to steer a virtual vehicle in a racing game by rotating the device as if it were a steering wheel.

The Android platform, based on the Linux kernel, is designed primarily for touchscreen mobile devices such as smartphones and tablets. Since Android devices are usually battery-powered, Android is designed to manage processes to keep power consumption at a minimum, providing longer battery use.

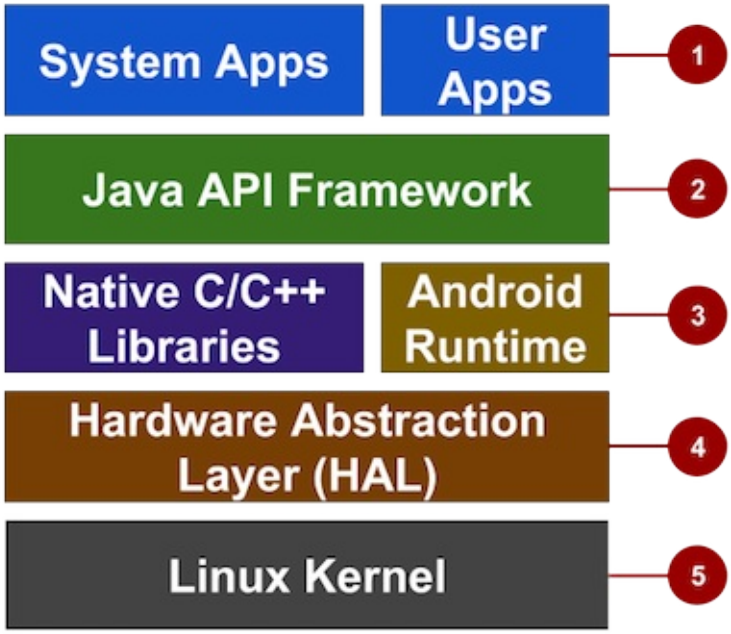
Easy to develop apps

Use the Android software development kit (SDK) to develop apps that take advantage of the Android operating system and UI. The SDK includes a comprehensive set of development tools including a debugger, software libraries of prewritten code, a device emulator, documentation, sample code, and tutorials. Use these tools to create apps that look great and take advantage of the hardware capabilities available on each device.

To develop apps using the SDK, use the Java programming language for developing the app and Extensible Markup Language (XML) files for describing data resources. By writing the code in Java and creating a single app binary, you will have an app that can run on both phone and tablet form factors. You can declare your UI in lightweight sets of XML resources, one set for parts of the UI that are common to all form factors, and other sets for features specific to phones or tablets. At runtime, Android applies the correct resource sets based on its screen size, density, locale, and so on.

To help you develop your apps efficiently, Google offers a full Java Integrated Development Environment (IDE) called Android Studio, with advanced features for developing, debugging, and packaging Android apps. Using Android Studio, you can develop on any available Android device, or create virtual devices that emulate any hardware configuration.

Android provides a rich development architecture. You don't need to know much about the components of this architecture, but it is useful to know what is available in the system for your app to use. The following diagram shows the major components of the Android *stack* — the operating system and development architecture.



In the figure above:

1. **Apps:** Your apps live at this level, along with core system apps for email, SMS messaging, calendars, Internet browsing, or contacts.
2. **Java API Framework:** All features of Android are available to developers through application programming interfaces (APIs) written in the Java language. You don't need to know the details of all of the APIs to learn how to develop Android apps, but you can learn more about the following APIs, which are useful for creating apps:
   - View System used to build an app's UI, including lists, buttons, and menus.
   - Resource Manager used to access to non-code resources such as localized strings, graphics, and layout files.
   - Notification Manager used to display custom alerts in the status bar.
   - Activity Manager that manages the lifecycle of apps.
   - Content Providers that enable apps to access data from other apps.

- All framework APIs that Android system apps use.

3. **Libraries and Android Runtime:** Each app runs in its own process and with its own instance of the Android Runtime, which enables multiple virtual machines on low-memory devices. Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language, including some Java 8 language features that the Java API framework uses. Many core Android system components and services are built from native code that require native libraries written in C and C++. These native libraries are available to apps through the Java API framework.
4. **Hardware Abstraction Layer (HAL):** This layer provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or bluetooth module.
5. **Linux Kernel:** The foundation of the Android platform is the Linux kernel. The above layers rely on the Linux kernel for underlying functionalities such as threading and low-level memory management. Using a Linux kernel enables Android to take advantage of key security features and allows device manufacturers to develop hardware drivers for a well-known kernel.

Many distribution options

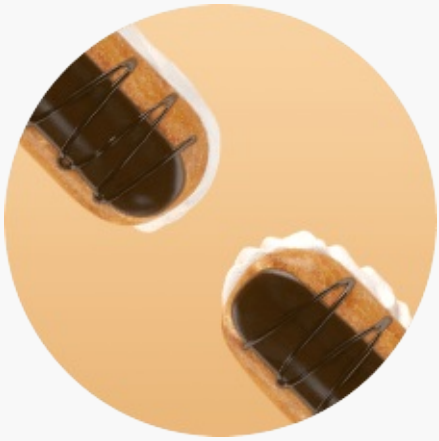
You can distribute your Android app in many different ways: email, website or an app marketplace such as Google Play. Android users download billions of apps and games from the Google Play store each month (shown in the figure below). Google Play is a digital distribution service, operated and developed by Google, that serves as the official appstore for Android, allowing consumers to browse and download apps developed with the Android SDK and published through

Google.

Android versions

Google provides major incremental upgrades to the Android operating system every six to nine months, using confectionery-themed names. The latest major release is Android 7.0 "Nougat".

Code name	Version number	Initial release date	API level
N/A	1.0	23 September 2008	1

N/A	1.1	9 February 2009	2
Cupcake	1.5	27 April 2009	3
Donut	1.6	15 September 2009	4
Eclair 	2.0 – 2.1	26 October 2009	5–7
Froyo	2.2 – 2.2.3	20 May 2010	8
Gingerbread			

	2.3 – 2.3.7	6 December 2010	9–10
Honeycomb	3.0 – 3.2.6	22 February 2011	11–13
Ice Cream Sandwich	4.0 – 4.0.4	18 October 2011	14–15
Jelly Bean			

	4.1 – 4.3.1	9 July 2012	16–18
KitKat	4.4 – 4.4.4	31 October 2013	19–20
Lollipop	5.0 – 5.1.1	12 November 2014	21–22
Marshmallow			

	6.0 – 6.0.1	5 October 2015	23
Nougat	7.0	22 August 2016	24

See previous versions and their features at The Android Story.

The Dashboard for Platform Versions is updated regularly to show the distribution of active devices running each version of Android, based on the number of devices that visit the Google Play Store. It's a good practice to support about 90% of the active devices, while targeting your app to the latest version.

Note: To provide the best features and functionality across Android versions, use the Android Support Library in your app. This support library allows your app to use recent platform APIs on older devices.

The challenges of Android app development

While the Android platform provide rich functionality for app development, there are still a number of challenges you need to address, such as:

- Building for a multi-screen world
- Getting performance right
- Keeping your code and your users secure
- Remaining compatible with older platform versions
- Understanding the market and the user.

Building for a multi-screen world

Android runs on billions of handheld devices around the world, and supports various form factors including wearable devices and televisions. Devices can come in different sizes and shapes that affect the screen designs for UI elements in your apps.

In addition, device manufacturers may add their own UI elements, styles, and colors to differentiate their products. Each manufacturer offers different features with respect to keyboard forms, screen size, or camera buttons. An app running on one device may look a bit different on another. The challenge for many developers is to design UI elements that can work on all devices It is also the developer's responsibility to provide an app's resources such as icons, logos, other graphics, and text styles to maintain uniformity of appearance across different devices.

Maximizing app performance

An app's performance—how fast it runs, how easily it connects to the network, and how well it manages battery and memory usage—is affected by factors such as battery life, multimedia content, and Internet access. You must be aware of these limitations and write code in such a way that the resource utilization is balanced and distributed optimally. For example, you will have to balance the background services by enabling them only when necessary; this will save battery life of the user's device.

Keeping your code and your users secure

You need to take precautions to secure your code and the user's experience when using your app. Use tools such as ProGuard (provided in Android Studio), which detects and removes unused classes, fields, methods, and attributes, and encrypt all of your app's code and resources while packaging the app. To protect your user's critical information such as logins and passwords, you must secure the communication channel to protect data in transit (across the Internet) as well as data at rest (on the device).

Remaining compatible with older platform versions

Consider how to add new Android platform version features to an app, while ensuring that the app can still run on devices with older platform versions. It is impractical to focus only on the most recent Android version, as not all users may have upgraded or may be able to upgrade their devices.

Learn more

- The Android Story
- Android API Guide, "Develop" section:
  - Introduction to Android
  - Platform Architecture

    - UI Overview
    - Platform Versions
    - Supporting Different Platform Versions
- Other:
    - Android Studio User's Guide: Image Asset Studio
    - Wikipedia: Summary of Android Version History

1.1: Create Your First Android App

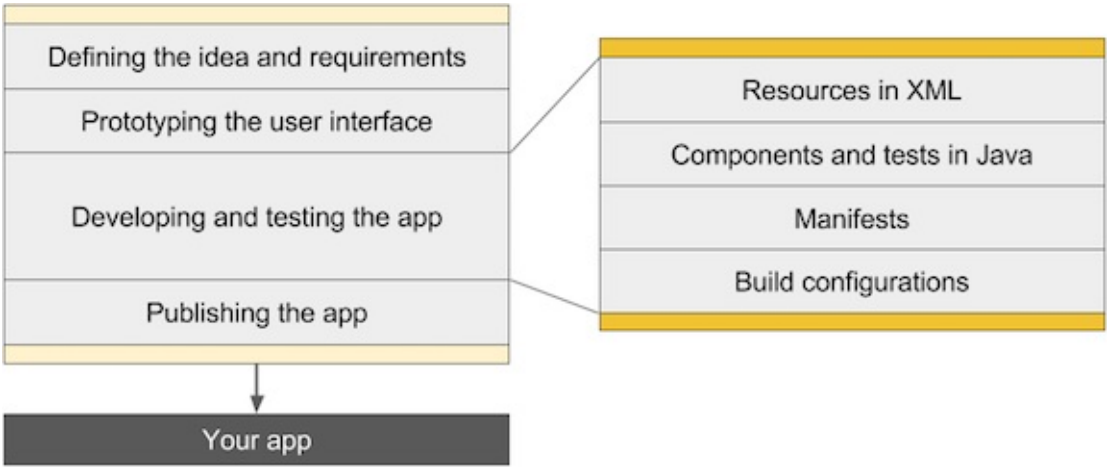
Contents:

- The development process
- Using Android Studio
- Exploring a project
- Viewing and editing Java code
- Viewing and editing layouts
- Understanding the build process
- Running the app on an emulator or a device
- Using the log
- Related practical
- Learn more

This chapter describes how to develop applications using the Android Studio Integrated Development Environment (IDE).

The development process

An Android app project begins with an idea and a definition of the requirements necessary to realize that idea. As the project progresses, it goes through design, development, and testing.



The above diagram is a high-level picture of the development process, with the following steps:

- *Defining the idea and its requirements:* Most apps start with an idea of what it should do, bolstered by market and user research. During this stage the app's requirements are defined.
- *Prototyping the user interface:* Use drawings, mock ups and prototypes to show what the user interface would look like, and how it would work.
- *Developing and testing the app:* An app consists of one or more activities. For each activity you can use Android Studio to do the following, in no particular order:
  - *Create the layout:* Place UI elements on the screen in a layout, and assign string resources and menu items, using the Extensible Markup Language (XML).
  - *Write the Java code:* Create source code for components and tests, and use testing and debugging tools.
  - *Register the activity:* Declare the activity in the manifest file.
  - *Define the build:* Use the default build configuration or create custom builds for different versions of your app.
- *Publishing the app*: Assemble the final APK (package file) and distribute it through channels such as the Google Play.

Using Android Studio

Android Studio provides tools for the testing, and publishing phases of the development process, and a unified development environment for creating apps for all Android devices. The development environment includes code templates with sample code for common app features, extensive testing tools and frameworks, and a flexible build system.

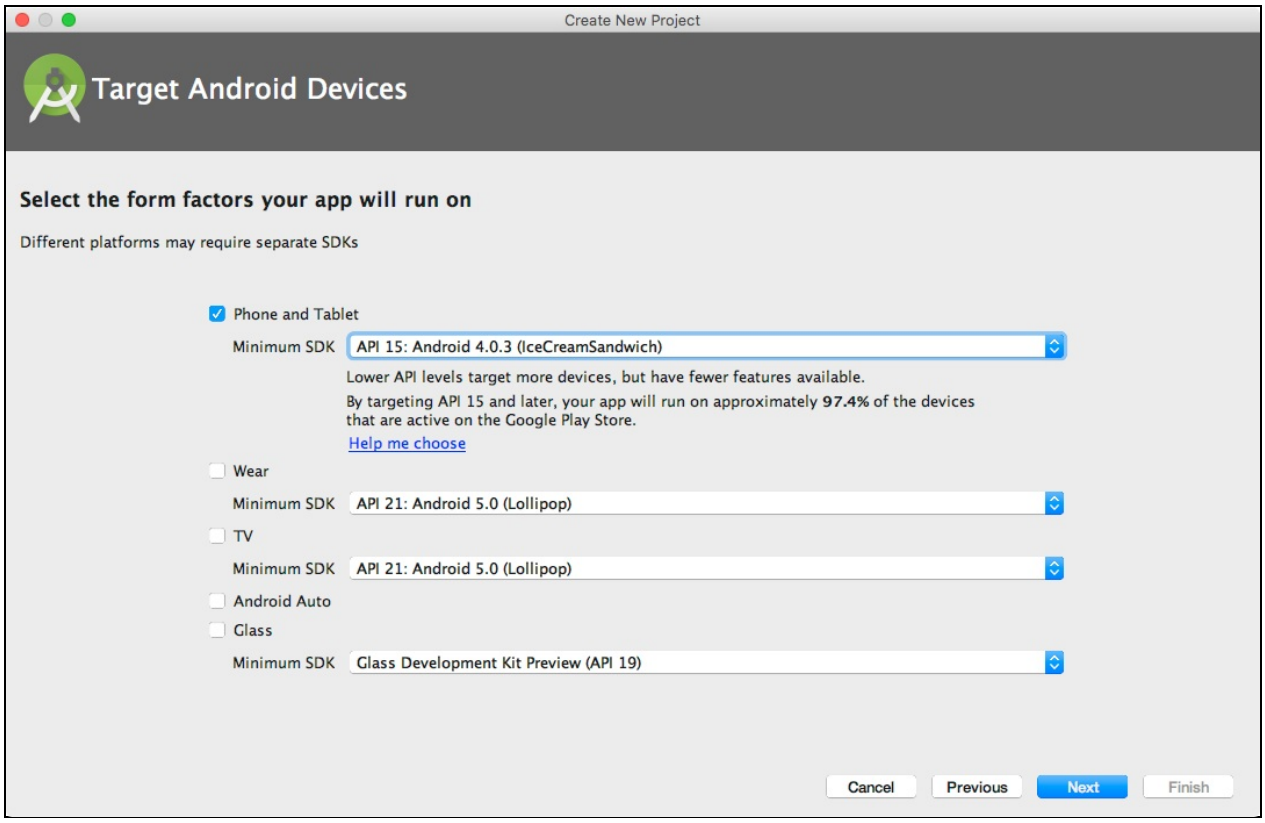
Starting an Android Studio project

After you have successfully installed the Android Studio IDE, double-click the Android Studio application icon to start it. Choose **Start a new Android Studio project** in the Welcome window, and name the project the same name that you want to use for the app.

When choosing a unique Company Domain, keep in mind that apps published to the Google Play must have a unique package name. Since domains are unique, prepending the app's name with your name, or your company's domain name, should provide an adequately unique package name. If you are not planning to publish the app, you can accept the default example domain. Be aware that changing the package name later is extra work.

Choosing target devices and the minimum SDK

When choosing Target Android Devices, Phone and Tablet are selected by default, as shown in the figure below. The choice shown in the figure for the Minimum SDK — **API 15: Android 4.0.3 (IceCreamSandwich)** — makes your app compatible with 97% of Android devices active on the Google Play Store.


Create New Project
Target Android Devices
Select the form factors your app will run on
Different platforms may require separate SDKs
Phone and Tablet
Minimum SDK
API 15: Android 4.0.3 (IceCreamSandwich)
Lower API levels target more devices, but have fewer features available.
By targeting API 15 and later, your app will run on approximately 97.4% of the devices that are active on the Google Play Store.
Help me choose
Wear
Minimum SDK
API 21: Android 5.0 (Lollipop)
TV
Minimum SDK
API 21: Android 5.0 (Lollipop)
Android Auto
Glass
Minimum SDK
Glass Development Kit Preview (API 19)
Cancel
Previous
Next
Finish

Different devices run different versions of the Android system, such as Android 4.0.3 or Android 4.4. Each successive version often adds new APIs not available in the previous version. To indicate which set of APIs are available, each version specifies an API level. For instance, Android 1.0 is API level 1 and Android 4.0.3 is API level 15.

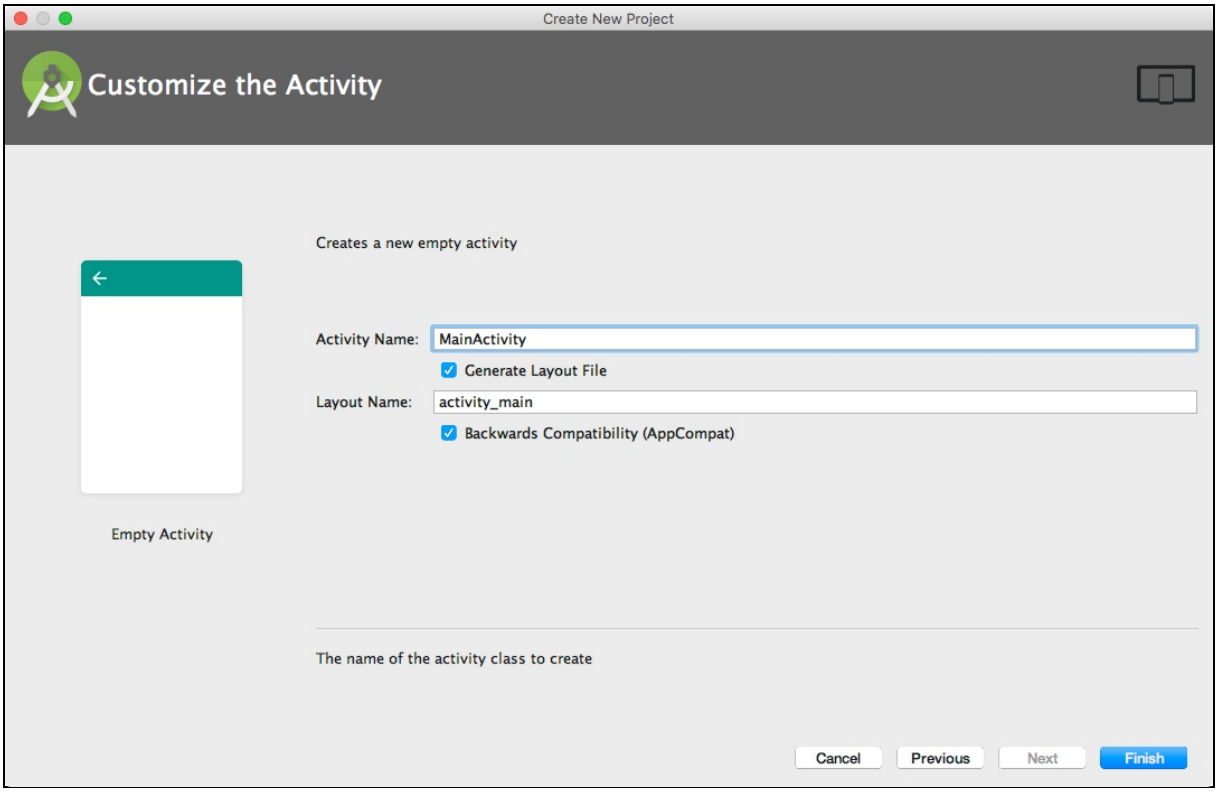
The Minimum SDK declares the minimum Android version for your app. Each successive version of Android provides compatibility for apps that were built using the APIs from previous versions, so your app should *always* be compatible with future versions of Android while using the documented Android APIs.

Choosing a template

Android Studio pre-populates your project with minimal code for an activity and a screen layout based on a *template*. A variety of templates are available, ranging from a virtually blank template (Add No Activity) to various types of activities.

You can customize the activity after choosing your template. For example, the Empty Activity template provides a single activity accompanied by a single layout resource for the screen. You can choose to accept the commonly used name for the activity (such as **MainActivity**) or change the name on the Customize the Activity screen. Also, if you use the Empty Activity template, be sure to check the following if they are not already checked:

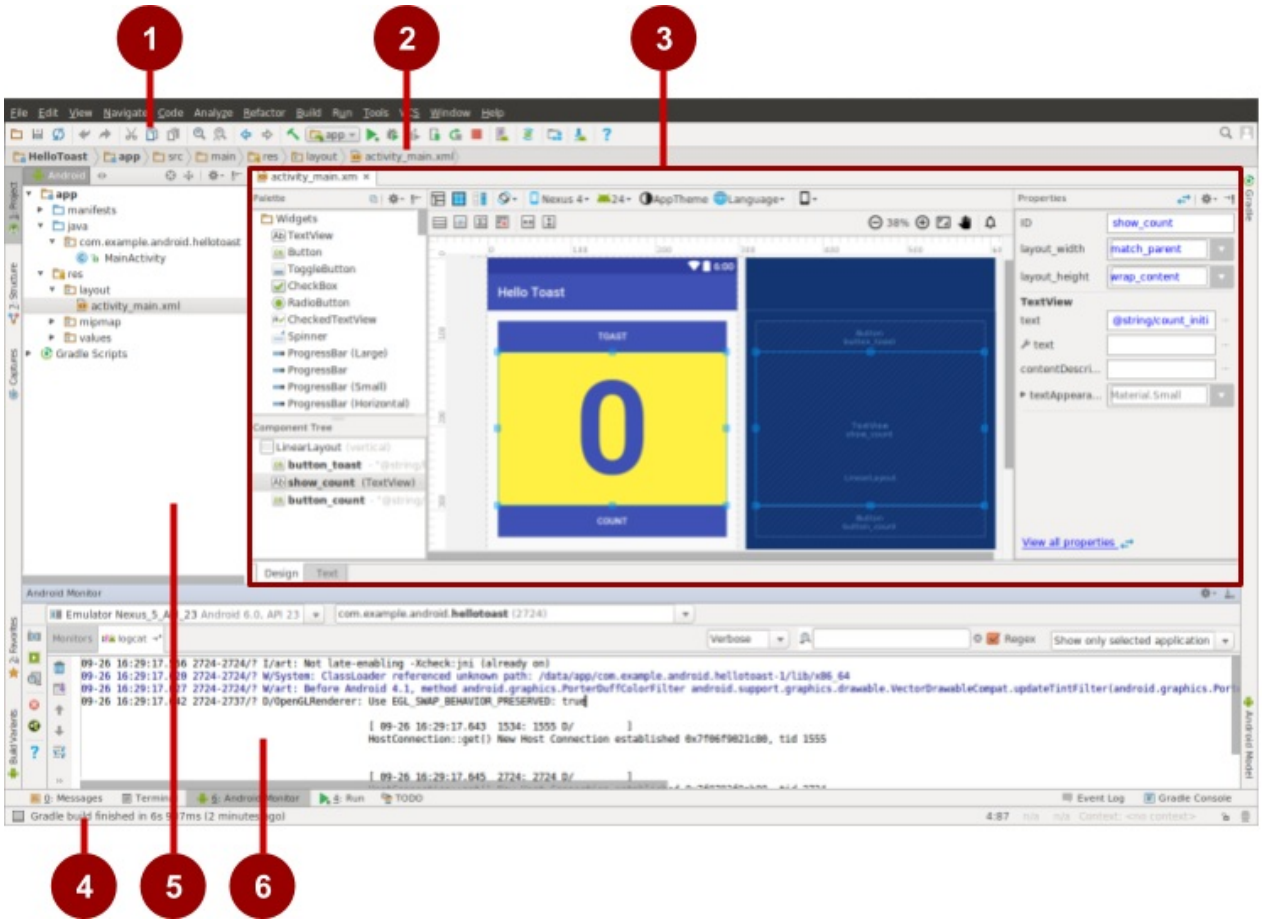
- Generate Layout file: Leave this checked to create the layout resource connected to this activity, which is usually named **activity_main.xml**. The layout defines the user interface for the activity.
- Backwards Compatibility (AppCompat)**:** Leave this checked to include the AppCompat library so that the app is compatible with previous versions of Android even if it uses features found only in newer versions.



Android Studio creates a folder for the newly created project in the AndroidStudioProjects folder on your computer.

Android Studio window panes

The Android Studio main window is made up of several logical areas, or *panes*, as shown in the figure below.



In the above figure:

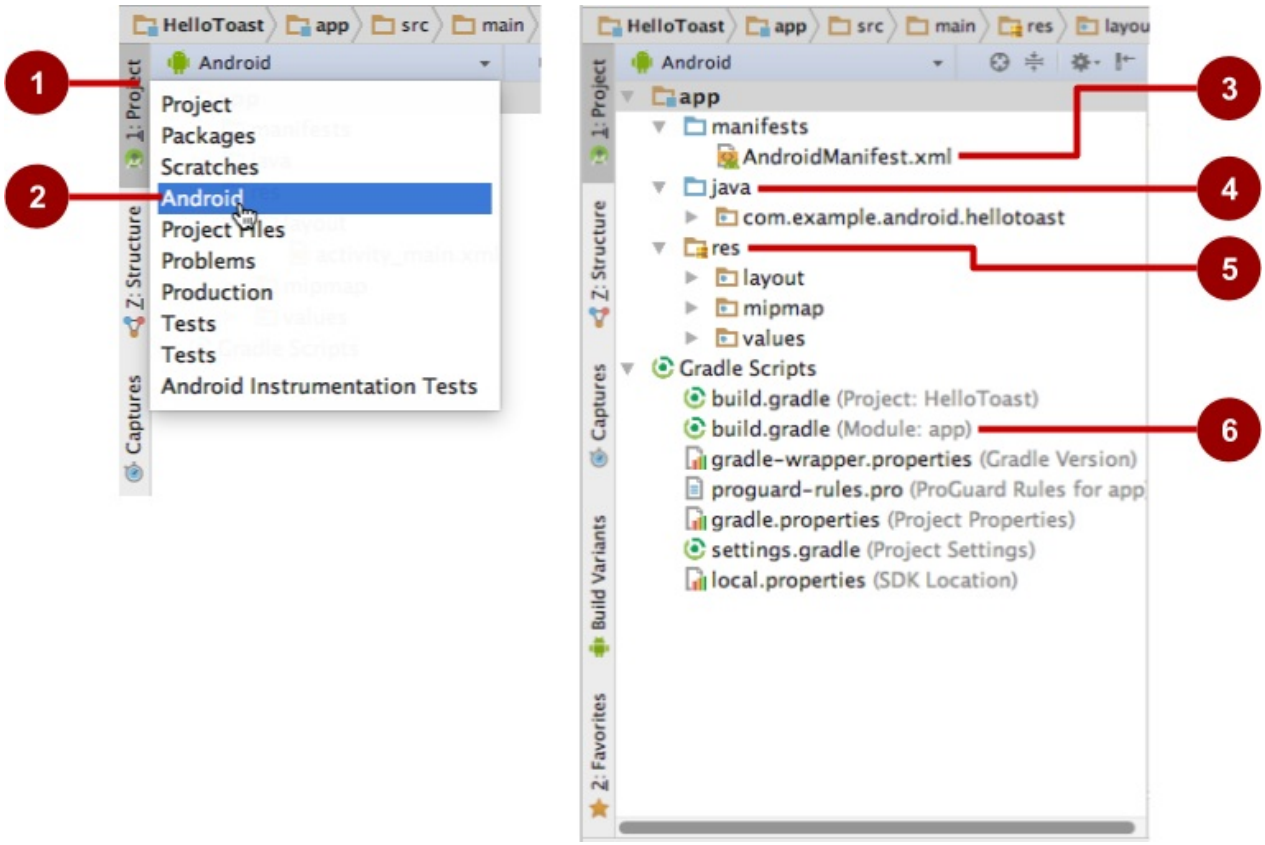
1. The **Toolbar**.The toolbar carries out a wide range of actions, including running the Android app and launching Android tools.
2. The **Navigation Bar**. The navigation bar allows navigation through the project and open files for editing. It provides a more compact view of the project structure.
3. The **Editor Pane**. This pane shows the contents of a selected file in the project. For example, after selecting a layout (as shown in the figure), this pane shows the layout editor with tools to edit the layout. After selecting a Java code file, this pane shows the code with tools for editing the code.
4. The **Status Bar.** The status bar displays the status of the project and Android Studio itself, as well as any warnings or messages. You can watch the build progress in the status bar.
5. The **Project Pane**. The project pane shows the project files and project hierarchy.
6. The **Monitor Pane**. The monitor pane offers access to the TODO list for managing tasks, the Android Monitor for monitoring app execution (shown in the figure), the logcat for viewing log messages, and the Terminal application for performing Terminal activities.

Tip: You can organize the main window to give yourself more screen space by hiding or moving panes. You can also use keyboard shortcuts to access most features. See Keyboard Shortcuts for a complete list.

Exploring a project

Each project in Android Studio contains the AndroidManifest.xml file, component source-code files, and associated resource files. By default, Android Studio organizes your project files based on the file type, and displays them within the Project: Android view in the left tool pane, as shown below. The view provides quick access to your project's key files.

To switch back to this view from another view, click the vertical **Project** tab in the far left column of the Project pane, and choose **Android** from the pop-up menu at the top of the Project pane, as shown in the figure below.



In the figure above:

1. The **Project** tab. Click to show the project view.
2. The **Android** selection in the project drop-down menu.
3. The **AndroidManifest.xml** file. Used for specifying information about the app for the Android runtime environment. The template you choose creates this file.
4. The **java** folder. This folder includes activities, tests, and other components in Java source code. Every activity, service, and other component is defined as a Java class, usually in its own file. The name of the first activity (screen) the user sees, which also initializes app-wide resources, is customarily MainActivity.
5. The **res** folder. This folder holds resources, such as XML layouts, UI strings, and images. An activity usually is associated with an XML resource file that specifies the layout of its views. This file is usually named after its activity or function.
6. The **build.gradle (Module: App)** file. This file specifies the module's build configuration. The template you choose creates this file, which defines the build configuration, including the `minSdkVersion` attribute that declares the minimum version for the app, and the `targetSdkVersion` attribute that declares the highest (newest) version for which the app has been optimized. This file also includes a list of *dependencies*, which are libraries required by the code — such as the AppCompat library for supporting a wide range of Android versions.

1

Viewing the Android Manifest

Before the Android system can start an app component, the system must know that the component exists by reading the app's AndroidManifest.xml file. The app must declare all its components in this file, which must be at the root of the app project directory.

To view this file, expand the manifests folder in the Project: Android view, and double-click the file (**AndroidManifest.xml**). Its contents appear in the editing pane as shown in the figure below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.helloworld">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Hello World"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Android namespace and application tag

The Android Manifest is coded in XML and always uses the Android namespace:

```
xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.android.helloworld">
```

The `package` expression shows the unique package name of the new app. Do not change this once the app is published.

```
   <application
   ...
   </application>
```

The `<application` tag, with its closing `</application>` tag, defines the manifest settings for the entire app.

Automatic backup

The `android:allowBackup` attribute enables automatic app data backup:

```
   ...
   android:allowBackup="true"
   ...
```

Setting the `android:allowBackup` attribute to `true` enables the app to be backed up automatically and restored as needed. Users invest time and effort to configure apps. Switching to a new device can cancel out all that careful configuration. The system performs this automatic backup for nearly all app data by default, and does so without the developer having to write any additional app code.

For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically create backups of app data to the cloud because the `android:allowBackup` attribute defaults to `true` if omitted. For apps < API level 22 you have to explicitly add the `android:allowBackup` attribute and set it to `true`.

Tip: To learn more about the automatic backup for apps, see Configuring Auto Backup for Apps.

The app icon

The `android:icon` attribute sets the icon for the app:

```
    ...
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    ...
```

The `android:icon` attribute assigns an icon in the **mipmap** folder (inside the **res** folder in Project: Android view) to the app. The icon appears in the Launcher for launching the app. The icon is also used as the default icon for app components.

App label and string resources

As you can see in the previous figure, the `android:label` attribute shows the string `"Hello World"` highlighted. If you click on this string, it changes to show the string resource `@string/app_name`:

```
    ...
    android:label="@string/app_name"
    ...
```

Tip: Ctrl-click or right-click `app_name` in the edit pane to see the context menu. Choose **Go To > Declaration** to see where the string resource is declared: in the strings.xml file. When you choose **Go To > Declaration** or open the file by double-clicking **strings.xml** in the Project: Android view (inside the **values** folder), its contents appear in the editing pane.

After opening the strings.xml file, you can see that the string name `app_name` is set to `Hello World`. You can change the app name by changing the `Hello World` string to something else. String resources are described in a separate lesson.

The app theme

The `android:theme` attribute sets the app's theme, which defines the appearance of user interface elements such as text:

```
    ...
    android:theme="@style/AppTheme">
    ...
```

The `theme` attribute is set to the standard theme `AppTheme`. Themes are described in a separate lesson.

Declaring the Android version

Different devices may run different versions of the Android system, such as Android 4.0 or Android 4.4. Each successive version can add new APIs not available in the previous version. To indicate which set of APIs are available, each version specifies an API level. For instance, Android 1.0 is API level 1 and Android 4.4 is API level 19.

The API level allows a developer to declare the minimum version with which the app is compatible, using the `<uses-sdk>` manifest tag and its `minSdkVersion` attribute. For example, the Calendar Provider APIs were added in Android 4.0 (API level 14). If your app can't function without these APIs, declare API level 14 as the app's minimum supported version like this:

```
<manifest ... >
    <uses-sdk android:minSdkVersion="14" android:targetSdkVersion="19" />
    ...
</manifest>
```

The `minSdkVersion` attribute declares the minimum version for the app, and the `targetSdkVersion` attribute declares the highest (newest) version which has been optimized within the app. Each successive version of Android provides compatibility for apps that were built using the APIs from previous versions, so the app should *always* be compatible with future versions of Android while using the documented Android APIs.

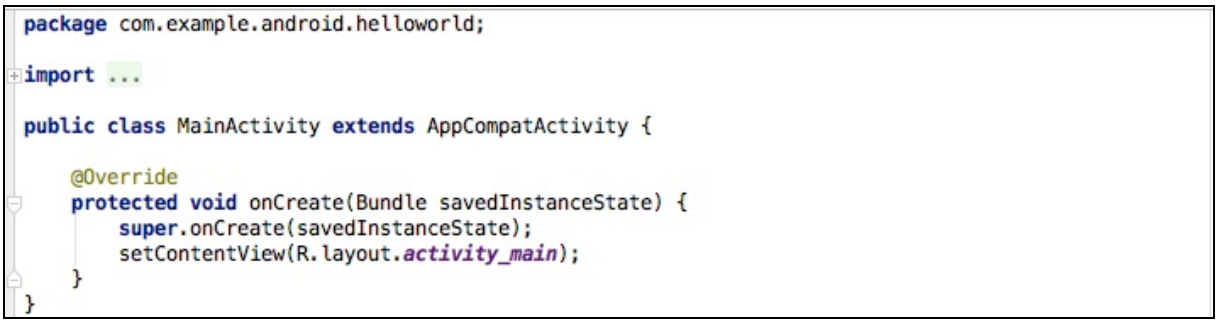
The `targetSdkVersion` attribute does *not* prevent an app from being installed on Android versions that are higher (newer) than the specified value, but it is important because it indicates to the system whether the app should inherit behavior changes in newer versions. If you don't update the `targetSdkVersion` to the latest version, the system assumes that your app requires some backward-compatibility behaviors when running on the latest version. For example, among the behavior changes in Android 4.4, alarms created with the AlarmManager APIs are now inexact by default so that the system can batch app alarms and preserve system power, but the system will retain the previous API behavior for an app if your target API level is lower than `"19"`.

Viewing and editing Java code

Components are written in Java and listed within module folders in the **java** folder in the Project: Android view. Each module name begins with the domain name (such as **com.example.android**) and includes the app name.

The following example shows an *activity* component:

1. Click the module folder to expand it and show the MainActivity file for the activity written in Java (the `MainActivity` class).
2. Double-click **MainActivity** to see the source file in the editing pane, as shown in the figure below.



```
package com.example.android.helloworld;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

At the very top of the MainActivity.java file is a `package` statement that defines the app package. This is followed by an `import` block condensed in the above figure, with " `...` ". Click the dots to expand the block to view it. The `import` statements import libraries needed for the app, such as the following, which imports the AppCompatActivity library:

```
import android.support.v7.app.AppCompatActivity;
```

Each activity in an app is implemented as a Java class. The following class declaration extends the `AppCompatActivity` class to implement features in a way that is backward-compatible with previous versions of Android:

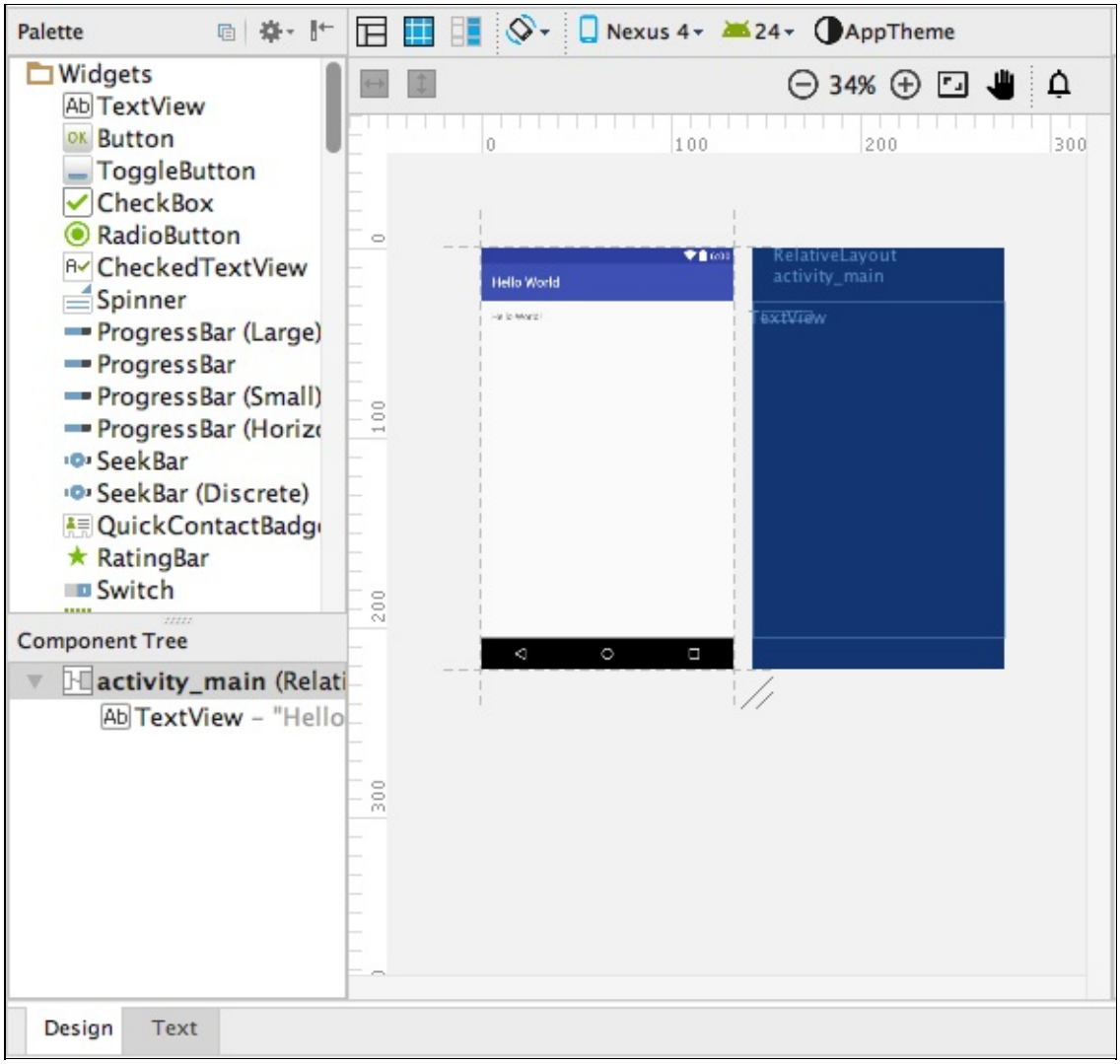
```
public class MainActivity extends AppCompatActivity {
    ...
}
```

As you learned earlier, before the Android system can start an app component such as an activity, the system must know that the activity exists by reading the app's AndroidManifest.xml file. Therefore, each activity must be listed in the AndroidManifest.xml file.

Viewing and editing layouts

Layout resources are written in XML and listed within the **layout** folder in the **res** folder in the Project: Android view. Click **res > layout** and then double-click **activity_main.xml** to see the layout file in the editing pane.

Android Studio shows the Design view of the layout, as shown in the figure below. This view provides a Palette pane of user interface elements, and a grid showing the screen layout.



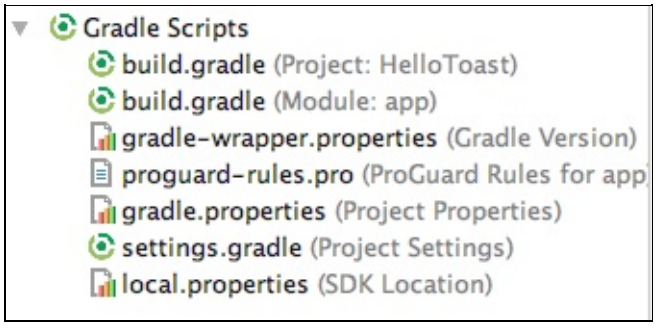
Understanding the build process

The Android application package (APK) is the package file format for distributing and installing Android mobile apps. The build process involves tools and processes that automatically convert each project into an APK.

Android Studio uses Gradle as the foundation of the build system, with more Android-specific capabilities provided by the Android Plugin for Gradle. This build system runs as an integrated tool from the Android Studio menu.

Understanding build.gradle files

When you create a project, Android Studio automatically generates the necessary build files in the **Gradle Scripts** folder in Project: Android view. Android Studio build files are named **build.gradle** as shown below:



Each project has the following:

build.gradle (Project: *apptitle*)

This is the top-level build file for the entire project, located in the root project directory, which defines build configurations that apply to all modules in your project. This file, generated by Android Studio, should not be edited to include app dependencies.

build.gradle (Module: app)

Android Studio creates separate `build.gradle (Module: app)` files for each module. You can edit the build settings to provide custom packaging options for each module, such as additional build types and product flavors, and to override settings in the manifest or top-level build.gradle file. This file is most often the file to edit when changing app-level configurations, such as declaring dependencies in the `dependencies` section. The following shows the contents of a project's `build.gradle (Module: app)` file:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
    buildToolsVersion "24.0.1"
    defaultConfig {
        applicationId "com.example.android.helloworld2"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
}
```

The build.gradle files use Gradle syntax. Gradle is a Domain Specific Language (DSL) for describing and manipulating the build logic using Groovy, which is a dynamic language for the Java Virtual Machine (JVM). You don't need to learn Groovy to make changes, because the Android Plugin for Gradle introduces most of the DSL elements you need.

Tip: To learn more about the Android plugin DSL, read the DSL reference documentation.

Plugin and Android blocks

In the above `build.gradle (Module: app)` file, the first statement applies the Android-specific Gradle plug-in build tasks:

```
apply plugin: 'com.android.application'

android {
   ...
}
```

The `android { }` block specifies the following for the build:

- The target SDK version for compiling the code:

```
compileSdkVersion 24
```

- The version of the build tools to use for building the app:

```
buildToolsVersion "24.0.1"
```

The defaultConfig block

Core settings and entries for the app are specified in a `defaultConfig { }` block within the `android { } block:`

```
   ...
   defaultConfig {
      applicationId "com.example.hello.helloworld"
      minSdkVersion 15
      targetSdkVersion 23
      versionCode 1
      versionName "1.0"
            testInstrumentationRunner
               "android.support.test.runner.AndroidJUnitRunner"
   }
...
```

The `minSdkVersion` and `targetSdkVersion` settings override any AndroidManifest.xml settings for the minimum SDK version and the target SDK version. See "Declaring the Android version" previously in this chapter for background information on these settings.

The `testInstrumentationRunner` statement adds the instrumentation support for testing the user interface with Espresso and UIAutomator. These are described in a separate lesson.

Build types

Build types for the app are specified in a `buildTypes { }` block, which controls how the app is built and packaged.

```
...
buildTypes {
   release {
      minifyEnabled false
      proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                            'proguard-rules.pro'
   }
}
...
```

The build type specified is `release` for the app's release. Another common build type is `debug`. Configuring build types is described in a separate lesson.

Dependencies

Dependencies for the app are defined in the `dependencies { }` block, which is the part of the build.gradle file that is most likely to change as you start developing code that depends on other libraries. The block is part of the standard Gradle API and belongs *outside* the `android { }` block.

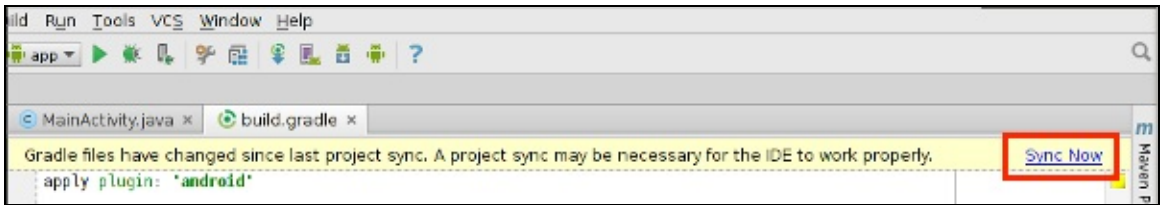
```
...
dependencies {
   compile fileTree(dir: 'libs', include: ['*.jar'])
   androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
exclude group: 'com.android.support', module: 'support-annotations'
   })
   compile 'com.android.support:appcompat-v7:24.2.0'
   testCompile 'junit:junit:4.12'
}
```

In the above snippet, the statement `compile fileTree(dir: 'libs', include: ['*.jar'])` adds a dependency of all ".jar" files inside the `libs` directory. The `compile` configuration compiles the main application — everything in it is added to the compilation classpath, and also packaged into the final APK.

Syncing your project

When you make changes to the build configuration files in a project, Android Studio requires that you *sync* the project files so that it can import the build configuration changes and run some checks to make sure the configuration won't create build errors.

To sync the project files, click **Sync Now** in the notification bar that appears when making a change, or click **Sync Project** from the menu bar. If Android Studio notices any errors with the configuration — for example, if the source code uses API features that are only available in an API level higher than the `compileSdkVersion` — the Messages window appears to describe the issue.


Run Tools VCS Window Help
app
MainActivity.java
build.gradle
Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.
Sync Now
apply plugin: 'android'

Running the app on an emulator or a device

With virtual device emulators, you can test an app on different devices such as tablets or smartphones — with different API levels for different Android versions — to make sure it looks good and works for most users. Although it's a good idea, you don't have to depend on having a physical device available for app development.

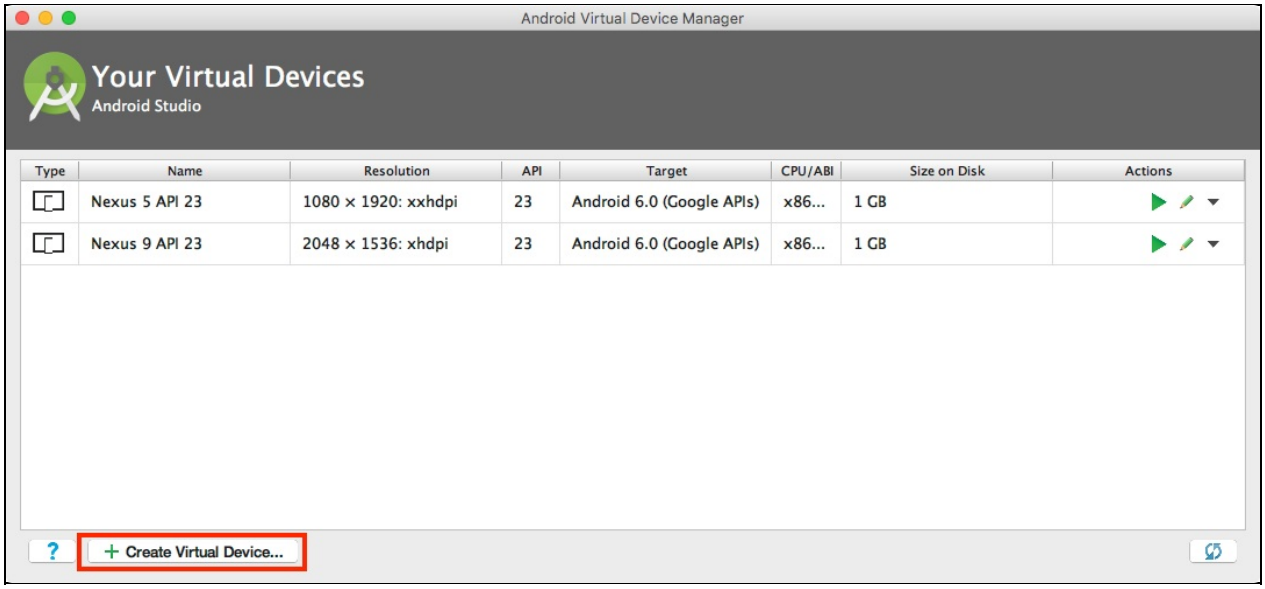
The Android Virtual Device (AVD) manager creates a virtual device or emulator that simulates the configuration for a particular type of Android device. Use the AVD Manager to define the hardware characteristics of a device and its API level, and to save it as a virtual device configuration. When you start the Android emulator, it reads a specified configuration and creates an emulated device on your computer that behaves exactly like a physical version of that device.

Creating a virtual device

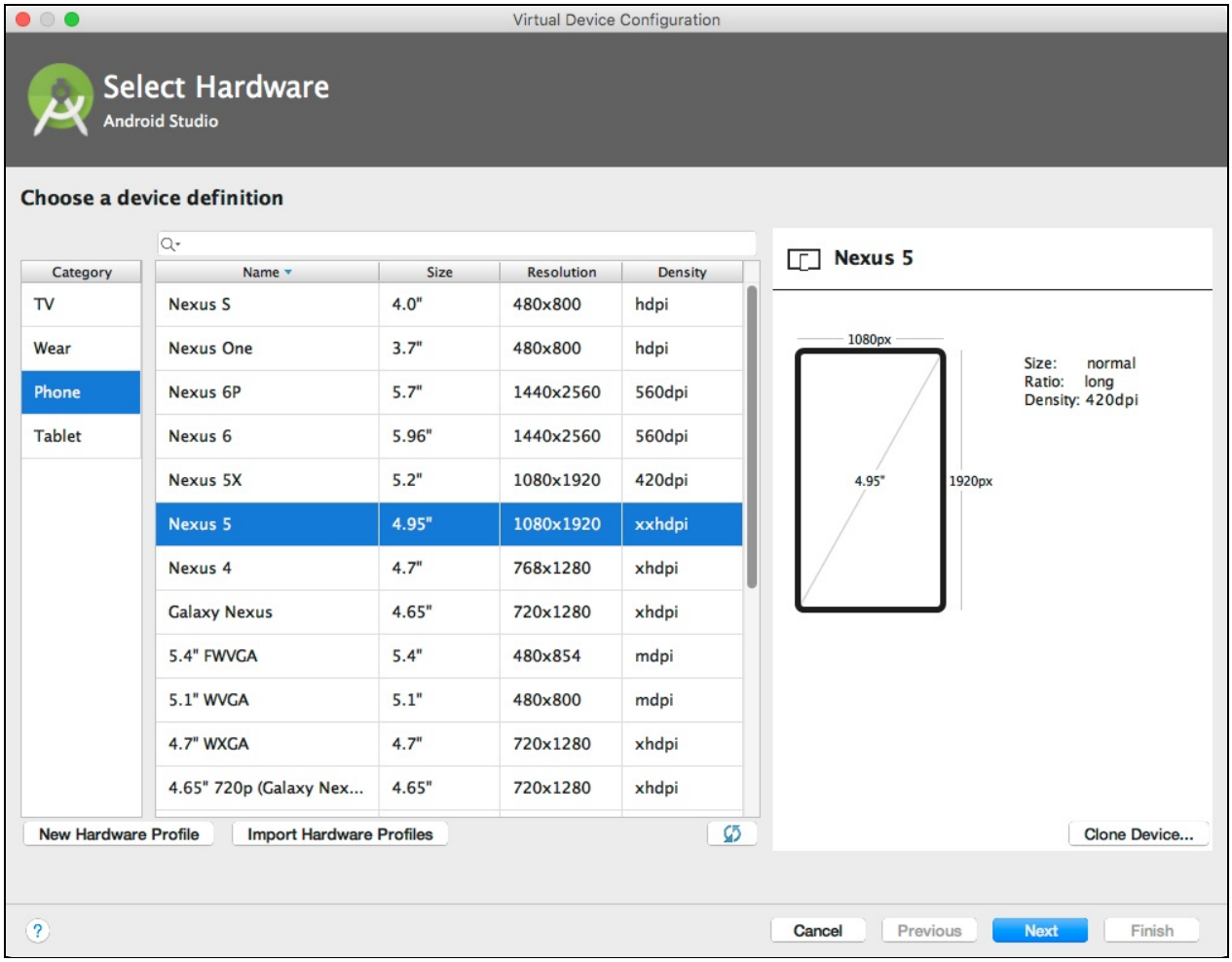
To run an emulator on your computer, use the AVD Manager to create a configuration that describes the virtual device.

Select **Tools > Android > AVD Manager**, or click the AVD Manager icon in the toolbar.

The "Your Virtual Devices" screen appears showing all of the virtual devices created previously. Click the **+Create Virtual Device** button to create a new virtual device.



You can select a device from a list of predefined hardware devices. For each device, the table shows its diagonal display size (Size), screen resolution in pixels (Resolution), and pixel density (Density). For example, the pixel density of the Nexus 5 device is `xxhdpi`, which means the app uses the icons in the **xxhdpi** folder of the **mipmap** folder. Likewise, the app will use layouts and drawables from folders defined for that density as well.



You also choose the version of the Android system for the device. The **Recommended** tab shows the recommended systems for the device. More versions are available under the **x86 Images** and **Other Images** tabs.

Running the app on the virtual device

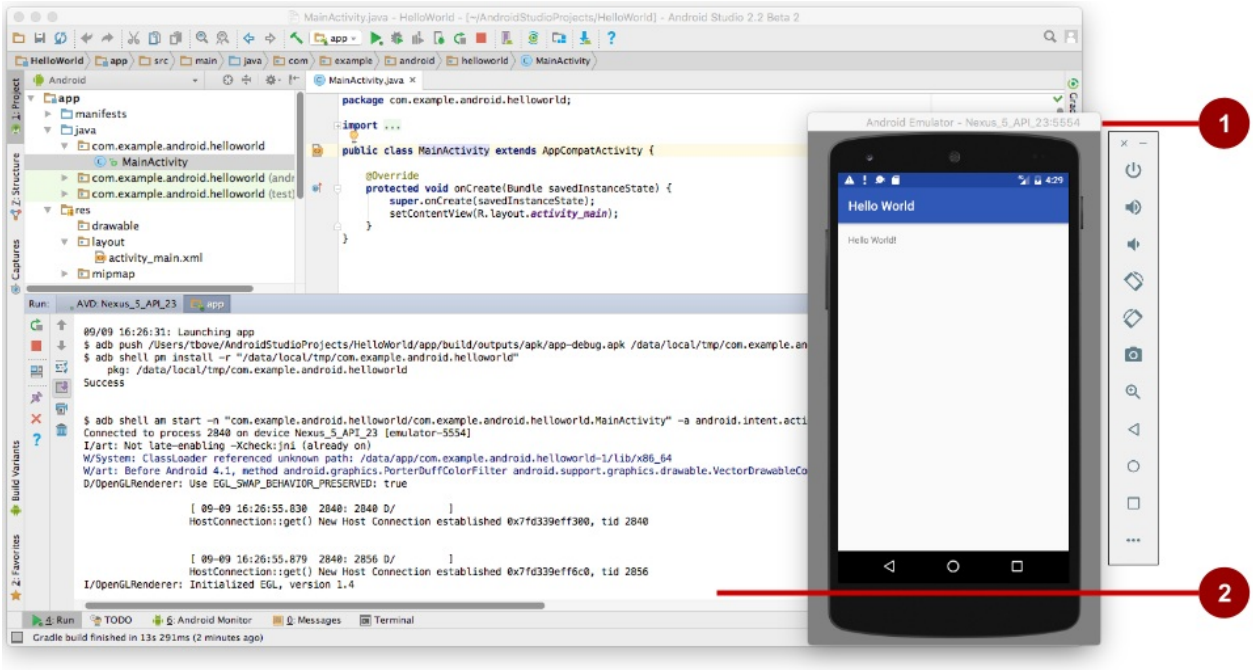
To run the app on the virtual device you created in the previous section, follow these steps:

1. In Android Studio, select **Run > Run app** or click the **Run icon** in the toolbar.
2. In the Select Deployment Target window, under Available Emulators, select the virtual device you created, and click **OK**.

The emulator starts and boots just like a physical device. Depending on the speed of your computer, this may take a while. The app builds, and once the emulator is ready, Android Studio uploads the app to the emulator and runs it.

You should see the app created from the Empty Activity template ("Hello World") as shown in the following figure, which also shows Android Studio's Run pane that displays the actions performed to run the app on the emulator.

Note: When testing on an emulator, it is good practice to start it up once at the very beginning of your session, and not to close it until done so that it doesn't have to go through the boot process again.



In the above figure:

1. The **Emulator** running the app.
2. The **Run Pane**. This shows the actions taken to install and run the app.

Running the app on a physical device

Always test your apps on physical device, because users will use the app on physical devices. While emulators are quite good, they can't show all possible device states, such as what happens if an incoming call occurs while the app is running. To run the app on a physical device, you need the following:

- An Android device such as a smartphone or tablet.
- A data cable to connect the Android device to your computer via the USB port.
- If you are using Linux or Windows, it may be necessary to perform additional steps to run the app on a hardware device. Check the Using Hardware Devices documentation. On Windows, you may need to install the appropriate USB driver for the device. See OEM USB Drivers.

To let Android Studio communicate with a device, turn on USB Debugging on the Android device. On Android version 4.2 and newer, the Developer options screen is hidden by default. Follow these steps to turn on USB Debugging:

1. On the physical device, open **Settings** and choose **About phone** at the bottom of the Settings screen.

2. Tap the **Build number** information seven times.

   You read that correctly: Tap it *seven times*.

3. Return to the previous screen (**Settings**). **Developer options** now appears at the bottom of the screen. Tap **Developer options**.
4. Choose **USB Debugging**.

Now, connect the device and run the app from Android Studio.

Troubleshooting the device connection

If Android Studio does not recognize the device, try the following:

1. Disconnect the device from your computer, and then reconnect it.
2. Restart Android Studio.
3. If your computer still does not find the device or declares it "unauthorized":

   i. Disconnect the device from your computer.

   ii. On the device, choose **Settings > Developer Options**.

   iii. Tap **Revoke USB Debugging authorizations**.

   iv. Reconnect the device to your computer.

   v. When prompted, grant authorizations.

4. You may need to install the appropriate USB driver for the device. See Using Hardware Devices documentation.
5. Check the latest documentation, programming forums, or get help from your instructors.

Using the log

The log is a powerful debugging tool you can use to look at values, execution paths, and exceptions. After you add logging statements to an app, your log messages appear along with general log messages in the logcat tab of the Android Monitor pane of Android Studio.

To see the Android Monitor pane, click the **Android Monito**r button at the bottom of the Android Studio main window. The Android Monitor offers two tabs:

- The **logcat** tab. The **logcat** tab displays log messages about the app as it is running. If you add logging statements to the app, your log messages from these statements appear with the other log messages under this tab.
- The **Monitors** tab. The **Monitors** tab monitors the performance of the app, which can be helpful for debugging and tuning your code.

Adding logging statements to your app

Logging statements add whatever messages you specify to the log. Adding logging statements at certain points in the code allows the developer to look at values, execution paths, and exceptions.

For example, the following logging statement adds `"MainActivity"` and `"Hello World"` to the log:

```
Log.d("MainActivity", "Hello World");
```

The following are the elements of this statement:

- `Log` : The Log class is the API for sending log messages.
- `d` : You assign a *log level* so that you can filter the log messages using the drop-down menu in the center of the **logcat** tab pane. The following are log levels you can assign:

- `d` : Choose **Debug** or **Verbose** to see these messages.
- `e` : Choose **Error** or **Verbose** to see these messages.
- `w` : Choose **Warning** or **Verbose** to see these messages.
- `i` : Choose **Info** or **Verbose** to see these messages.

- `"MainActivity"` : The first argument is a *log tag* which can be used to filter messages under the **logcat** tab. This is commonly the name of the activity from which the message originates. However, you can make this anything that is useful to you for debugging the app. The best practice is to use a constant as a log tag, as follows:

1. Define the log tag as a constant before using it in logging statement:

```
private static final String LOG_TAG =
     MainActivity.class.getSimpleName();
```

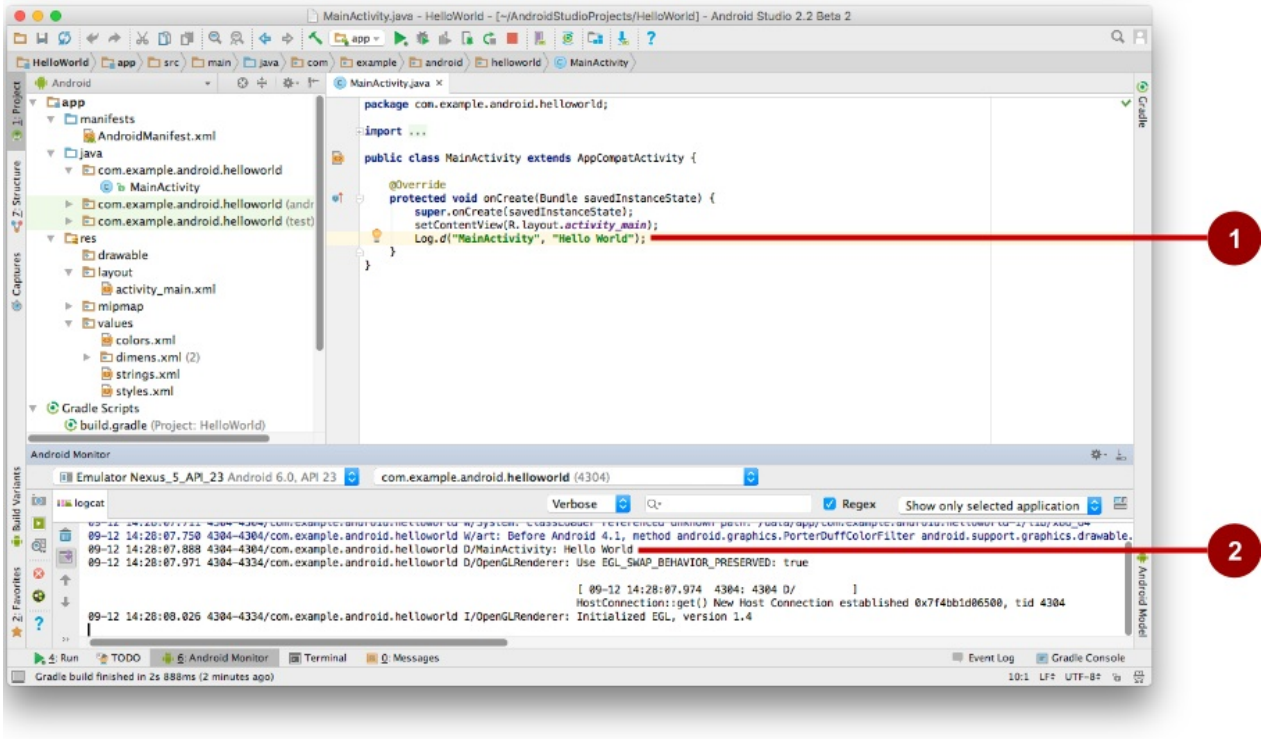
2. Use the constant in the logging statements:

```
Log.d(LOG_TAG, "Hello World");
```

3. `"Hello World"` : The second argument is the actual message that appears after the log level and log tag under the **logcat** tab.

Viewing your log messages

The Run pane appears in place of the Android Monitor pane when you run the app on an emulator or a device. After starting to run the app, click the **Android Monitor** button at the bottom of the main window, and then click the **logcat** tab in the Android Monitor pane if it is not already selected.



In the above figure:

1. The logging statement in the `onCreate()` method of `MainActivity` .
2. Android Monitor pane showing `logcat` log messages, including the message from the logging statement.

By default, the log display is set to **Verbose** in the drop-down menu at the top of the **logcat** display to show all messages. You can change this to **Debug** to see messages that start with `Log.d` , or change it to **Error** to see messages that start with `Log.e` , and so on.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Install Android Studio and Run "Hello World"

Learn more

- Android Studio documentation:
  - Meet Android Studio
  - Android Studio download page
  - Configure Build Variants
  - Create and Manage Virtual Devices
  - Sign Your App
  - Shrink Your Code and Resources
- Android API Guide, "Develop" section:
  - Introduction to Android
  - Platform Architecture
  - UI Overview
  - Platform Versions
  - Supporting Different Platform Versions
  - Supporting Multiple Screens
- Other:
  - Android Studio User's Guide: Image Asset Studio
  - Wikipedia: Summary of Android Version History
  - Groovy syntax
  - Gradle site

1.2: Layouts, Views and Resources

Contents:

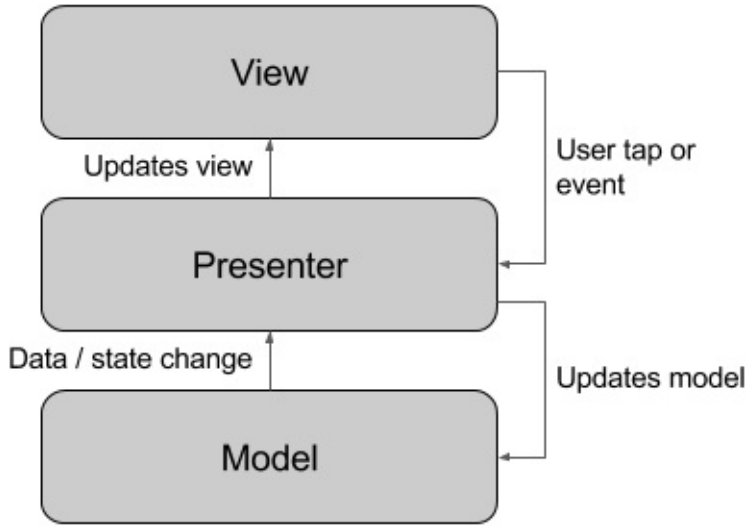
- The model-view-presenter pattern
- Views
- Resource files
- Responding to view clicks
- Related practical
- Learn more

This chapter describes the screen's user interface layout and other resources you create for your app, and the code you would use to respond to a user's tap of a user interface element.

The model-view-presenter pattern

Linking an activity to a layout resource is an example of part of the *model-view-presenter* (MVP) architecture pattern. The MVP pattern is a well-established way to group app functions:

- **Views.** Views are user interface elements that display data and respond to user actions. Every element of the screen is a view. The Android system provides many different kinds of views.
- **Presenters.** Presenters connect the application's views to the model. They supply the views with data as specified by the model, and also provide the model with user input from the view.
- **Model.** The model specifies the structure of the app's data and the code to access and manipulate the data. Some of the apps you create in the lessons work with models for accessing data. The Hello Toast app does not use a data model, but you can think of its logic — display a message, and increase a tap counter — as the model.



Views

The UI consists of a hierarchy of objects called *views* — every element of the screen is a *view*. The View class represents the basic building block for all UI components, and the base class for classes that provide interactive UI components such as buttons, checkboxes, and text entry fields.

A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the device-independent pixel (dp).

The Android system provides hundreds of predefined views, including those that display:

- Text (TextView)
- Fields for entering and editing text (EditText)
- Buttons users can tap (Button) and other interactive components
- Scrollable text (ScrollView) and scrollable items (RecyclerView)
- Images (ImageView)

You can define a view to appear on the screen and respond to a user tap. A view can also be defined to accept text input, or to be invisible until needed.

You can specify the views in XML layout resource files. Layout resources are written in XML and listed within the **layout** folder in the **res** folder in the Project: Android view.

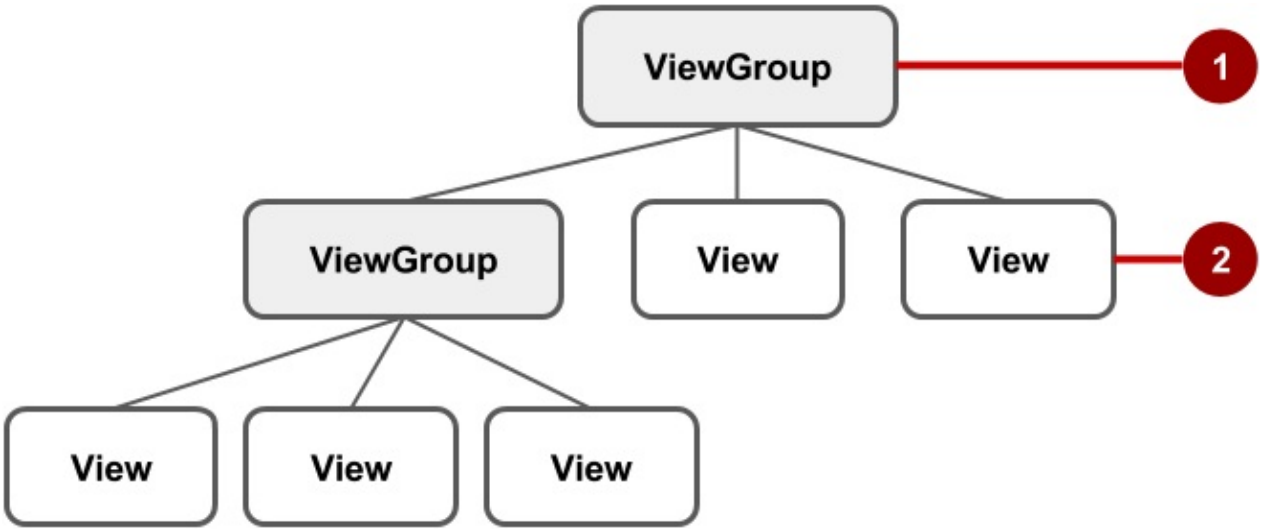
View groups

Views can be grouped together inside a *view group* (ViewGroup), which acts as a container of views. The relationship is parent-child, in which the *parent* is a view group, and the *child* is a view or view group within the group. The following are common view groups:

- ScrollView: A group that contains one other child view and enables scrolling the child view.
- RecyclerView: A group that contains a list of other views or view groups and enables scrolling them by adding and removing views dynamically from the screen.

Layout view groups

The views for a screen are organized in a hierarchy. At the *root* of this hierarchy is a ViewGroup that contains the layout of the entire screen. The view group's child screens can be other views or other view groups as shown in the following figure.



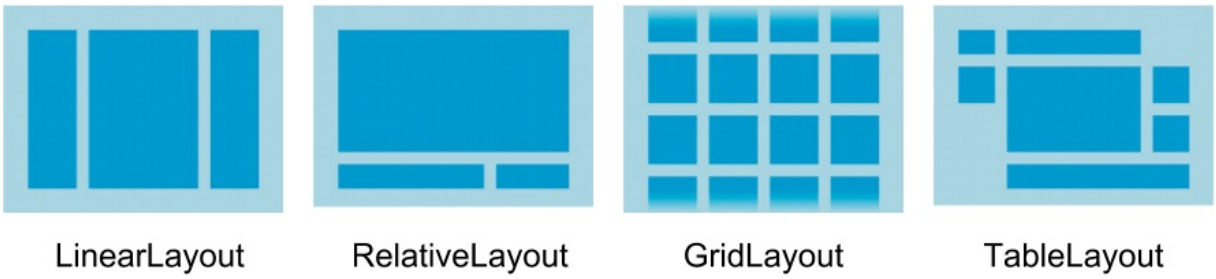
In the above figure:

1. The *root* view group.
2. The first set of child views and view groups whose parent is the root.

Some view groups are designated as *layouts* because they organize child views in a specific way and are typically used as the root view group. Some examples of layouts are:

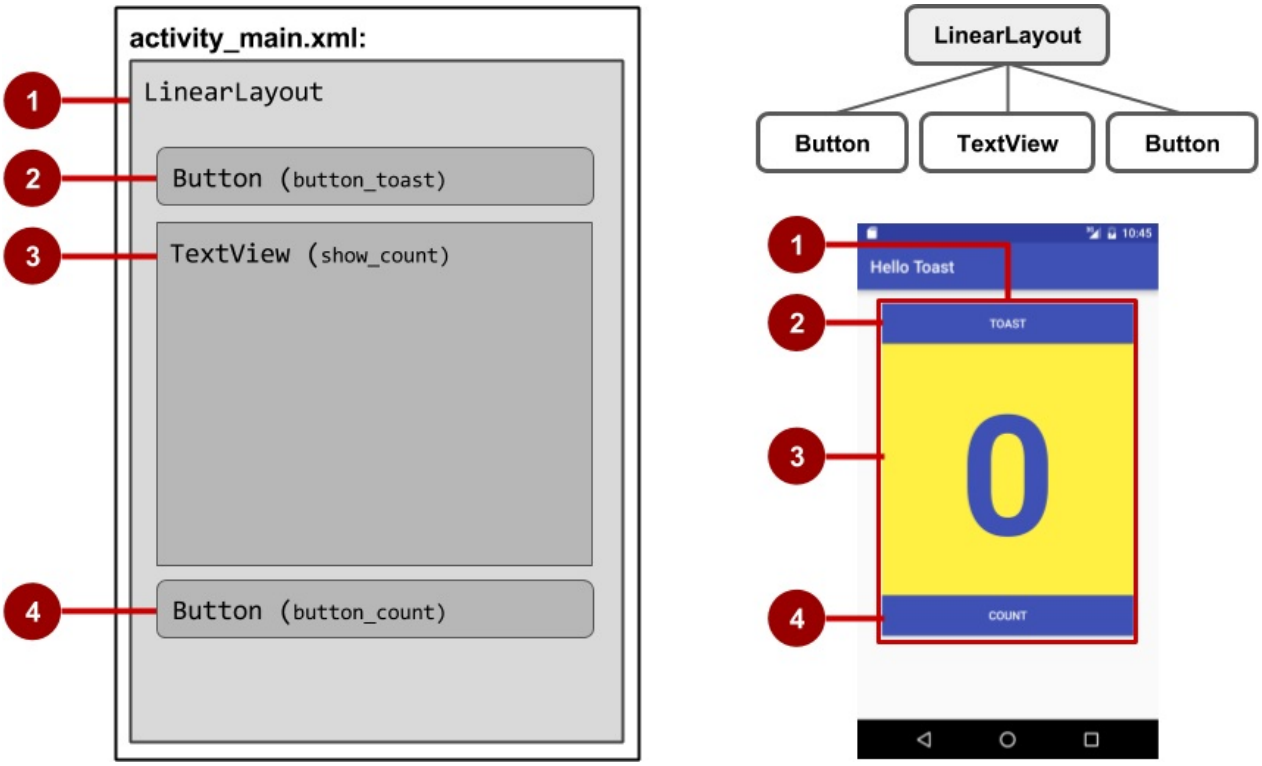
- LinearLayout: A group of child views positioned and aligned horizontally or vertically.
- RelativeLayout: A group of child views in which each view is positioned and aligned relative to other views within the view group. In other words, the positions of the child views can be described in relation to each other or to the parent view group.
- ConstraintLayout: A group of child views using anchor points, edges, and guidelines to control how views are positioned relative to other elements in the layout. ConstraintLayout was designed to make it easy to drag and drop views in the layout editor.

- TableLayout: A group of child views arranged into rows and columns.
- AbsoluteLayout: A group that lets you specify exact locations (x/y coordinates) of its child views. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.
- FrameLayout: A group of child views in a stack. FrameLayout is designed to block out an area on the screen to display one view. Child views are drawn in a stack, with the most recently added child on top. The size of the FrameLayout is the size of its largest child view.
- GridLayout: A group that places its child screens in a rectangular grid that can be scrolled.



Tip: Learn more about different layout types in Common Layout Objects.

A simple example of a layout with child views is the Hello Toast app in one of the early lessons. The view for the Hello Toast app appears in the figure below as a diagram of the layout file (activity_main.xml), along with a hierarchy diagram (top right) and a screenshot of the actual finished layout (bottom right).



In the figure above:

1. LinearLayout root layout, which contains all the child views, set to a vertical orientation.
2. Button (`button_toast`) child view. As the first child view, it appears at the top in the linear layout.
3. TextView (`show_count`) child view. As the second child view, it appears under the first child view in the linear layout.
4. Button (`button_count`) child view. As the third child view, it appears under the second child view in the linear layout.

The view hierarchy can grow to be complex for an app that shows many views on a screen. It's important to understand the view hierarchy, as it affects whether views are visible and efficiently they are drawn.

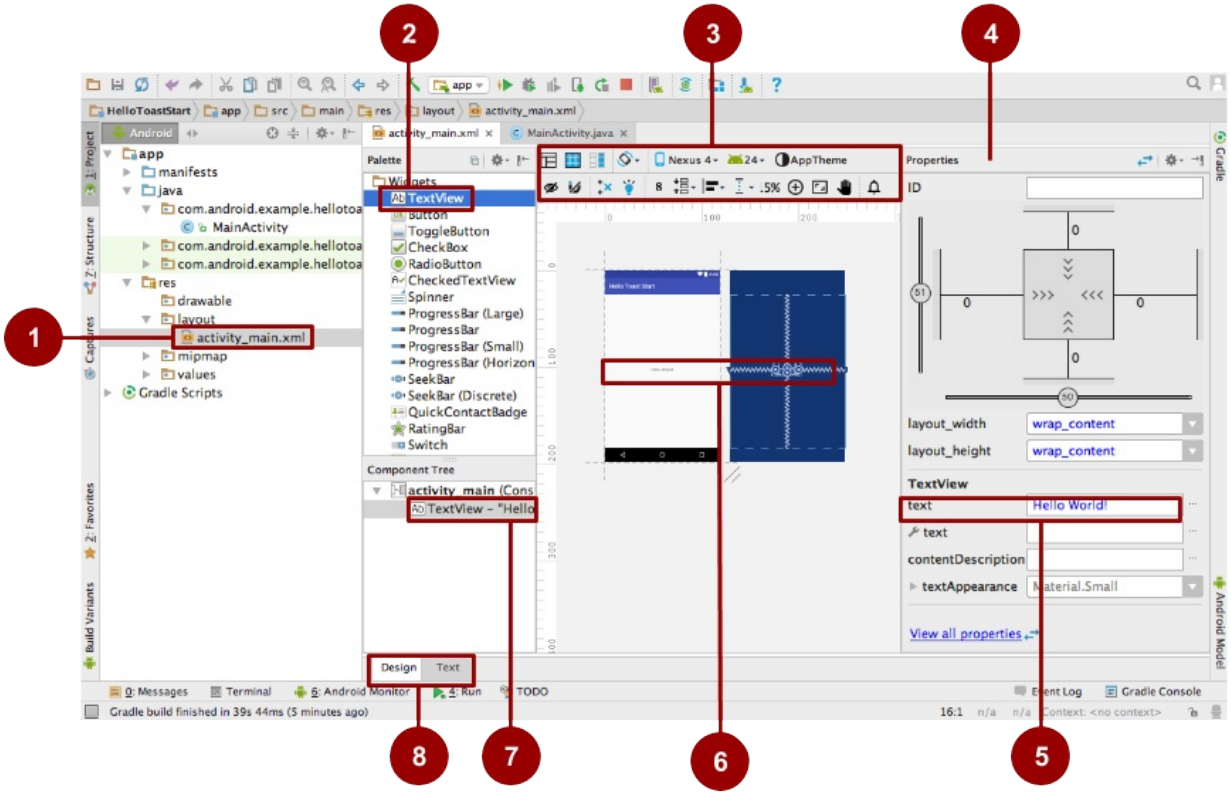
Tip: You can explore the view hierarchy of your app using Hierarchy Viewer. It shows a tree view of the hierarchy and lets you analyze the performance of views on an Android device. Performance issues are covered in a subsequent chapter.

You define views in the layout editor, or by entering XML code. The layout editor shows a visual representation of XML code.

Using the layout editor

Use the layout editor to edit the layout file. You can drag and drop view objects into a graphical pane, and arrange, resize, and specify properties for them. You immediately see the effect of changes you make.

To use the layout editor, open the XML layout file. The layout editor appears with the **Design** tab at the bottom highlighted. (If the **Text** tab is highlighted and you see XML code, click the **Design** tab.) For the Empty Activity template, the layout is as shown in the figure below.



In the figure above:

1. **XML layout** file. The XML layout file, typically named **activiy_main.xml** file. Double-click it to open the layout editor.
2. **Palette of UI elements** (views). The Palette pane provides a list of UI elements and layouts. Add an element or layout to the UI by dragging it into the design pane.
3. **Design toolbar**. The design pane toolbar provides buttons to configure your layout appearance in the design pane and to edit the layout properties. See the figure below for details.

   Tip: Hover over each icon to view a tooltip that summarizes its function.

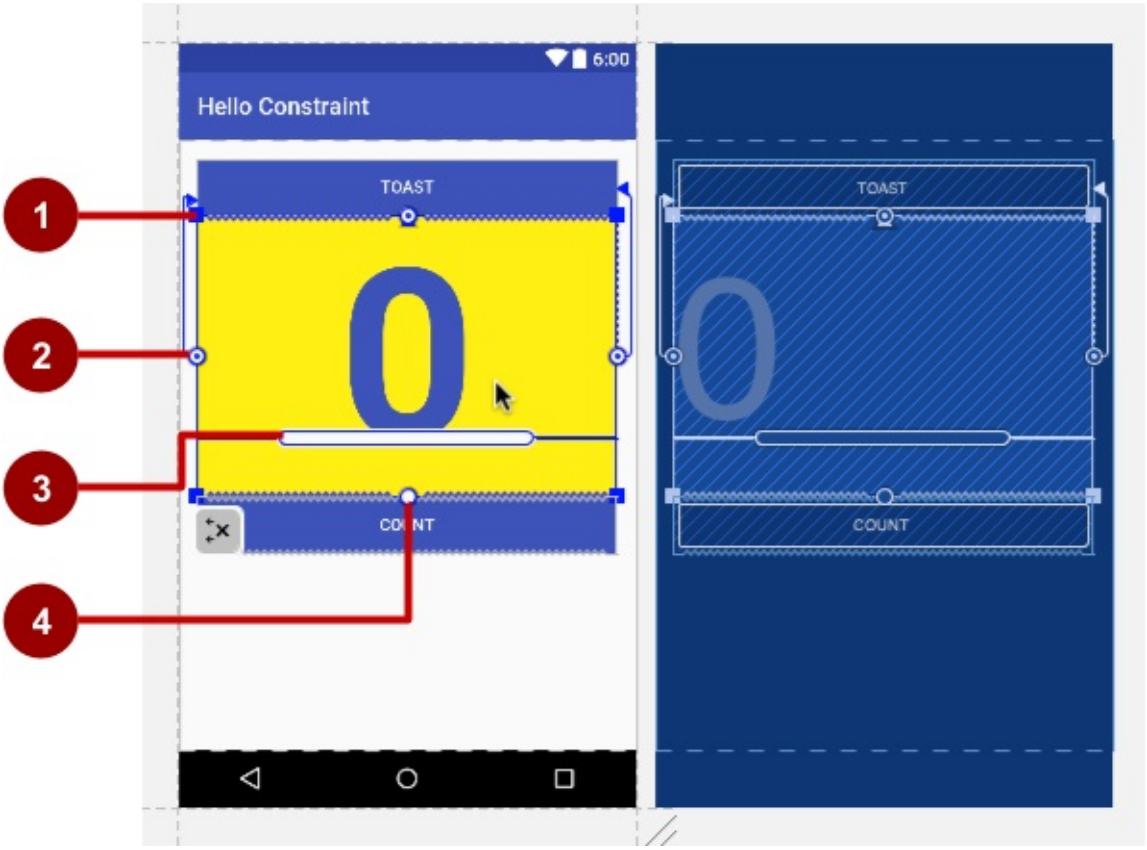
4. **Properties pane**. The Properties pane provides property controls for the selected view.
5. **Property control**. Property controls correspond to XML attributes. Shown in the figure is the `Text` property of the selected TextView, set to `Hello World!` .
6. **Design pane**. Drag views from the Palette pane to the design pane to position them in the layout.
7. **Component Tree**. The Component Tree pane shows the view hierarchy. Click a view or view group in this pane to select it. The figure shows the TextView selected.
8. **Design** and **Text** tabs. Click **Design** to see the layout editor, or **Text** to see XML code.

The layout editor's design toolbar offers a row of buttons that let you configure the appearance of the layout:

In the figure above:

1. **Design**, **Blueprint**, and **Both**: Click the **Design** icon (first icon) to display a color preview of your layout. Click the **Blueprint** icon (middle icon) to show only outlines for each view. You can see *both* views side by side by clicking the third icon.
2. **Screen orientation**: Click to rotate the device between landscape and portrait.
3. **Device type and size**: Select the device type (phone/tablet, Android TV, or Android Wear) and screen configuration (size and density).
4. **API version**: Select the version of Android on which to preview the layout.
5. **App theme**: Select which UI theme to apply to the preview.
6. **Language**: Select the language to show for your UI strings. This list displays only the languages available in the string resources.
7. **Layout Variants**: Switch to one of the alternative layouts for this file, or create a new one.

The layout editor offers more features in the **Design** tab when you use a ConstraintLayout, including handles for defining constraints. A *constraint* is a connection or alignment to another view, to the parent layout, or to an invisible guideline. Each constraint appears as a line extending from a circular handle. Each view has a circular constraint handle in the middle of each side. After selecting a view in the Component Tree pane or clicking on it in the layout, the view also shows resizing handles on each corner.



In the above figure:

1. **Resizing handle**.
2. **Constraint line and handle**. In the figure, the constraint aligns the left side of the view to the left side of the button.
3. **Baseline handle**. The baseline handle aligns the text baseline of a view to the text baseline of another view.
4. **Constraint handle** without a constraint line.

Using XML

It is sometimes quicker and easier to edit the XML code directly, especially when copying and pasting the code for similar views.

To view and edit the XML code, open the XML layout file. The layout editor appears with the **Design** tab at the bottom highlighted. Click the **Text** tab to see the XML code. The following shows an XML code snippet of a LinearLayout with a Button and a TextView:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ... >

    <Button
        android:id="@+id/button_toast"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="wrap_content"
        ... />

    <TextView
        android:id="@+id/show_count"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="@dimen/counter_height"
        ... />
    ...
</LinearLayout>
```

XML attributes (view properties)

Views have *properties* that define where a view appears on the screen, its size, how the view relates to other views, and how it responds to user input. When defining views in XML, the properties are referred to as *attributes*.

For example, in the following XML description of a TextView, the `android:id`, `android:layout_width`, `android:layout_height`, `android:background`, are XML attributes that are translated automatically into the TextView's properties:

```
<TextView
       android:id="@+id/show_count"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:background="@color/myBackgroundColor"
       android:textStyle="bold"
       android:text="@string/count_initial_value"
/>
```

Attributes generally take this form:

```
android:attribute_name="value"
```

The *attribute_name* is the name of the attribute. The *value* is a string with the value for the attribute. For example:

```
android:textStyle="bold"
```

If the *value* is a resource, such as a color, the `@` symbol specifies what kind of resource. For example:

```
android:background="@color/myBackgroundColor"
```

The background attribute is set to the color resource identified as `myBackgroundColor`, which is declared to be `#FFF043`. Color resources are described in "Style-related attributes" in this chapter.

Every view and view group supports its own variety of XML attributes. Some attributes are specific to a view (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any views that may extend the TextView class. Some are common to all views, because they are inherited from the root View class (like the `android:id` attribute). For descriptions of specific attributes, see the overview section of the View class documentation.

Identifying a view

To uniquely identify a view and reference it from your code, you must give it an id. The `android:id` attribute lets you specify a unique `id` — a resource identifier for a view.

For example:

```
android:id="@+id/button_count"
```

The `"@+id/button_count"` part of the above attribute creates a new `id` called `button_count` for the view. You use the plus (`+`) symbol to indicate that you are creating a new `id`.

Referencing a view

To refer to an existing resource identifier, omit the plus (`+`) symbol. For example, to refer to a view by its `id` in *another* attribute, such as `android:layout_toLeftOf` (described in the next section) to control the position of a view, you would use:

```
android:layout_toLeftOf="@id/show_count"
```

In the above attribute, `"@id/show_count"` refers to the view with the resource identifier `show_count`. The attribute positions the view to be "to the left of" the `show_count` view.

Positioning views

Some layout-related positioning attributes are required for a view, and automatically appear when you add the view to the XML layout, ready for you to add values.

LinearLayout positioning

For example, LinearLayout is required to have these attributes set:

- android:layout_width
- android:layout_height
- android:orientation

The `android:layout_width` and `android:layout_height` attributes can take one of three values:

- `match_parent` expands the view to fill its parent by width or height. When the LinearLayout is the root view, it expands to the size of the device screen. For a view within a root view group, it expands to the size of the parent view group.
- `wrap_content` shrinks the view dimensions just big enough to enclose its content. (If there is no content, the view becomes invisible.)
- Use a fixed number of `dp` (device-independent pixels) to specify a fixed size, adjusted for the screen size of the device. For example, `16dp` means 16 device-independent pixels. Device-independent pixels and other dimensions are described in "Dimensions" in this chapter.

The `android:orientation` can be:

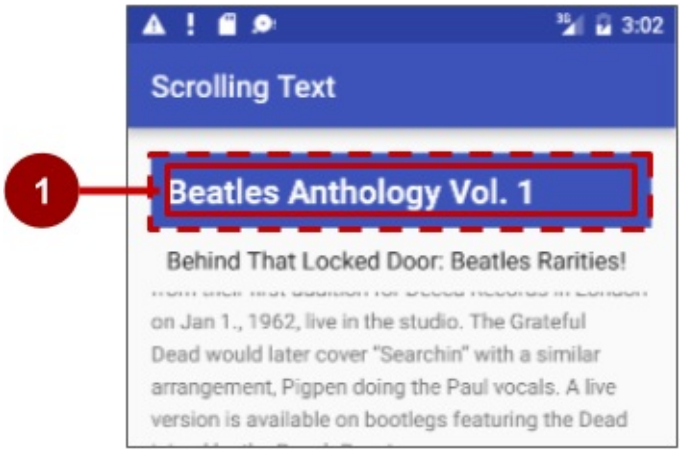
- `horizontal:` Views are arranged from left to right.
- `vertical:` Views are arranged from top to bottom.

Other layout-related attributes include:

- `Android:layout_gravity` : This attribute is used with a view to control where the view is arranged within its parent view group. For example, the following attribute centers the view horizontally on the screen:

  ```
  android:layout_gravity="center_horizontal"
  ```

- Padding is the space, measured in device-independent pixels, between the edges of the view and the view's content, as shown in the figure below.


3:02
Scrolling Text
1
Beatles Anthology Vol. 1
Behind That Locked Door: Beatles Rarities!
on Jan 1., 1962, live in the studio. The Grateful
Dead would later cover "Searchin" with a similar
arrangement, Pigpen doing the Paul vocals. A live
version is available on bootlegs featuring the Dead

In the figure above:

1. Padding is the space between the edges of the view (dashed lines) and the view's content (solid line). Padding is not the same as margin, which is the space from the edge of the view to its parent.

A view's size includes its padding. The following are commonly used padding attributes:

- `Android:padding` : Sets the padding of all four edges.
- `android:paddingTop` : Sets the padding of the top edge.
- `android:paddingBottom` : Sets the padding of the bottom edge.
- `android:paddingLeft` : Sets the padding of the left edge.
- `android:paddingRight` : Sets the padding of the right edge.
- `android:paddingStart` : Sets the padding of the start of the view; used in place of the above, especially with views that are long and narrow.
- `android:paddingEnd` : Sets the padding, in pixels, of the end edge; used along with `android:paddingStart` .

Tip: To see all of the XML attributes for a LinearLayout, see the Summary section of the LinearLayout reference in the Developer Guide. Other root layouts, such as RelativeLayout and AbsoluteLayout, list their XML attributes in the Summary sections.

RelativeLayout Positioning

Another useful view group for layout is RelativeLayout, which you can use to position child views relative to each other or to the parent. The attributes you can use with RelativeLayout include the following:

- android:layout_toLeftOf: Positions the right edge of this view to the left of another view (identified by its `ID`).
- android:layout_toRightOf: Positions the left edge of this view to the right of another view (identified by its `ID`).
- android:layout_centerHorizontal: Centers this view horizontally within its parent.
- android:layout_centerVertical: Centers this view vertically within its parent.
- android:layout_alignParentTop: Positions the top edge of this view to match the top edge of the parent.
- android:layout_alignParentBottom: Positions the bottom edge of this view to match the bottom edge of the parent.

For a complete list of attributes for views in a RelativeLayout, see RelativeLayout.LayoutParams.

Style-related attributes

You specify style attributes to customize the view's appearance. Views that *don't* have style attributes, such as `android:textColor`, `android:textSize`, and `android:background`, take on the styles defined in the app's theme.

The following are style-related attributes used in the XML layout example in the previous section:

- `Android:background`: Specifies a color or drawable resource to use as the background.
- `android:text`: Specifies text to display in the view.
- `android:textColor`: Specifies the text color.
- `android:textSize`: Specifies the text size.
- `android:textStyle`: Specifies the text style, such as `bold`.

Resource files

Resource files are a way of separating static values from code so that you don't have to change the code itself to change the values. You can store all the strings, layouts, dimensions, colors, styles, and menu text separately in resource files.

Resource files are stored in folders located in the **res** folder, including:

- **drawable**: For images and icons
- **layout**: For layout resource files
- **menu**: For menu items
- **mipmap**: For pre-calculated, optimized collections of app icons used by the Launcher
- **values**: For colors, dimensions, strings, and styles (theme attributes)

The syntax to reference a resource in an XML layout is as follows:

```
@package_name:resource_type/resource_name
```

- The *package_name* is the name of the package in which the resource is located. This is not required when referencing resources from the same package — that is, stored in the **res** folder of your project.
- *resource_type* is the `R` subclass for the resource type. See Resource Types for more information about each resource type and how to reference them.
- *resource_name* is either the resource filename without the extension, or the `android:name` attribute value in the XML element.

For example, the following XML layout statement sets the `android:text` attribute to a `string` resource:

```
android:text="@string/button_label_toast"
```

- The *resource_type* is `string`.
- The resource_name is `button_label_toast.`
- There is no need for a *package_name* because the resource is stored in the project (in the strings.xml file).

Another example: this XML layout statement sets the `android:background` attribute to a `color` resource, and since the resource is defined in the project (in the colors.xml file), the *package_name* is not specified:

```
android:background="@color/colorPrimary"
```

In the following example, the XML layout statement sets the `android:textColor` attribute to a `color` resource. However, the resource is not defined in the project but supplied by Android, so the *package_name* `android` must also be specified, followed by a colon:

```
android:textColor="@android:color/white"
```

Tip: For more information about accessing resources from code, see Accessing Resources. For Android color constants, see the Android standard R.color resources.

Values resource files

Keeping values such as strings and colors in separate resource files makes it easier to manage them, especially if you use them more than once in your layouts.

For example, it is essential to keep strings in a separate resource file for translating and localizing your app, so that you can create a string resource file for each language without changing your code. Resource files for images, colors, dimensions, and other attributes are handy for developing an app for different device screen sizes and orientations.

Strings

String resources are located in the **strings.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly by opening it:

```
<resources>
    <string name="app_name">Hello Toast</string>
    <string name="button_label_count">Count</string>
    <string name="button_label_toast">Toast</string>
    <string name="count_initial_value">0</string>
</resources>
```

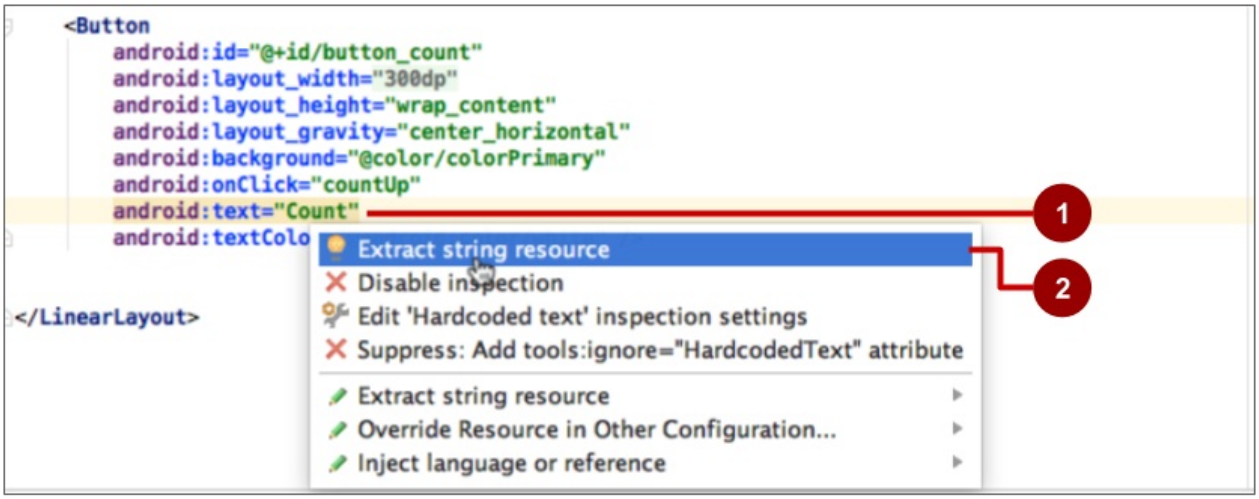
The `name` (for example, `button_label_count`) is the resource name you use in your XML code, as in the following attribute:

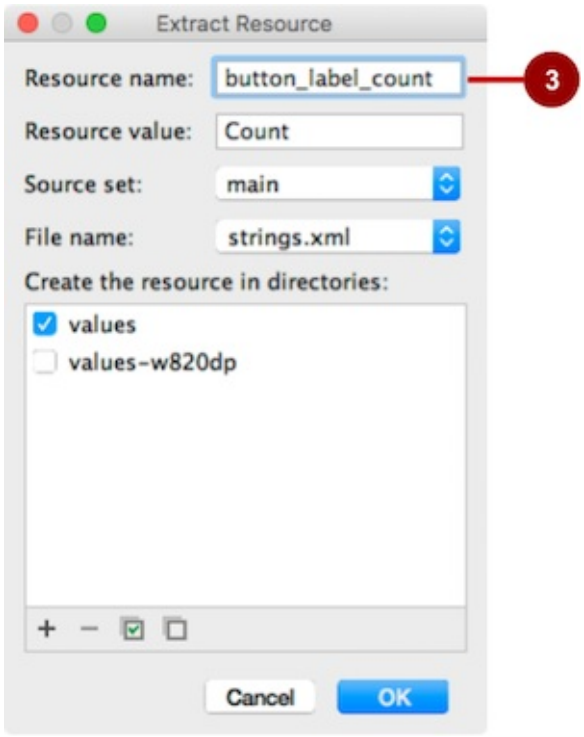
```
android:text="@string/button_label_count"
```

The string value of this `name` is the word (`Count`) enclosed within the `<string></string>` tags (you don't use quotation marks unless the quotation marks should be part of the string value.)

Extracting strings to resources

You should also *extract* hard-coded strings in an XML layout file to string resources. To extract a hard-coded string in an XML layout, follow these steps (refer to the figure):





1. Click on the hard-coded string, and press Alt-Enter in Windows, or Option-Return on Mac OS X.
2. Select **Extract string resource**.
3. Edit the Resource name for the string value.

You can then use the resource name in your XML code. Use the expression `"@string/resource_name"` (including quotation marks) to refer to the string resource:

```
android:text="@string/button_label_count"
```

Colors

Color resources are located in the **colors.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly:

```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
    <color name="myBackgroundColor">#FFF043</color>
</resources>
```

The `name` (for example, `colorPrimary`) is the resource name you use in your XML code:

```
android:textColor="@color/colorPrimary"
```

The color value of this `name` is the hexadecimal color value (`#3F51B5`) enclosed within the `<color></color>` tags. The hexadecimal value specifies red, green, and blue (RGB) values. The value always begins with a pound (`#`) character, followed by the Alpha-Red-Green-Blue information. For example, the hexadecimal value for black is #000000, while the hexadecimal value for a variant of sky blue is #559fe3. Base color values are listed in the Color class documentation.

The `colorPrimary` color is one of the predefined base colors and is used for the app bar. In a production app, you could, for example, customize this to fit your brand. Using the base colors for other UI elements creates a uniform UI.

Tip: For the material design specification for Android colors, see Style and Using the Material Theme. For common color hexadecimal values, see Color Hex Color Codes. For Android color constants, see the Android standard R.color resources.

You can see a small block of the color choice in the left margin next to the color resource declaration in **colors.xml**, and also in the left margin next to the attribute that uses the resource name in the layout XML file.

Tip: To see the color in a popup, turn on the Autopopup documentation feature. Choose **Android Studio > Preferences > Editor > General > Code Completion**, and check the "Autopopup documentation in (ms)" option. You can then hover your cursor over a color resource name to see the color.

Dimensions

Dimensions should be separated from the code to make them easier to manage, especially if you need to adjust your layout for different device resolutions. It also makes it easy to have consistent sizing for views, and to change the size of multiple views by changing one dimension resource.

Dimension resources are located in a **dimens.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. The **dimens.xml** shown in the view can be a folder holding more than one **dimens.xml** file for different device resolutions. For example, the app created from the Empty Activity template provides a second **dimens.xml** file for 820dp.

You can edit this file directly by opening it:

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
    <dimen name="my_view_width">300dp</dimen>
    <dimen name="count_text_size">200sp</dimen>
    <dimen name="counter_height">300dp</dimen>
</resources>
```

The `name` (for example, `activity_horizontal_margin`) is the resource name you use in the XML code:

```
android:paddingLeft="@dimen/activity_horizontal_margin"
```

The value of this `name` is the measurement (`16dp`) enclosed within the `<dimen></dimen>` tags.

You can extract dimensions in the same way as strings::

1. Click on the hard-coded dimension, and press Alt-Enter in Windows, or press Option-Return on Mac OS X.
2. Select **Extract dimension resource**.
3. Edit the Resource name for the dimension value.

Device-independent pixels (`dp`) are independent of screen resolution. For example, `10px` (10 fixed pixels) will look a lot smaller on a higher resolution screen, but Android will scale 1 `0dp` (10 device-independent pixels) to look right on different resolution screens. Text sizes can also be set to look right on different resolution screens using *scaled-pixel* (`sp`) sizes.

Tip: For more information about `dp` and `sp` units, see Supporting Different Densities.

Styles

A style is a resource that specifies common attributes such as height, padding, font color, font size, background color. Styles are meant for attributes that modify the look of the view.

Styles are defined in the **styles.xml** file in the **values** folder inside the **res** folder when using the Project: Android view. You can edit this file directly. Styles are covered in a later chapter, along with the Material Design Specification.

Other resource files

Android Studio defines other resources that are covered in other chapters:

- **Images and icons**. The **drawable** folder provides icon and image resources. If your app does not have a drawable folder, you can manually create it inside the res folder. For more information about drawable resources, see Drawable Resources in the App Resources section of the Android Developer Guide.
- **Optimized icons**. The **mipmap** folder typically contains pre-calculated, optimized collections of app icons used by the Launcher. Expand the folder to see that versions of icons are stored as resources for different screen densities.
- **Menus**. You can use an XML resource file to define menu items and store them in your project in the **menu** folder. Menus are described in a later chapter.

Responding to view clicks

A *click event* occurs when the user taps or clicks a clickable view, such as a Button, ImageButton, ImageView (tapping or clicking the image), or FloatingActionButton.

The model-view-presenter pattern is useful for understanding how to respond to view clicks. When an event occurs with a view, the *presenter* code performs an action that affects the *model* code. In order to make this pattern work, you have to:

- Write a Java method that performs the specific action, which is determined by the logic of the *model* code — that is, the action depends on what you want the app to do when this event occurs. This is typically referred to as an *event handler*.
- Associate this event handler method to the *view*, so that the method executes when the event occurs.

The onClick attribute

Android Studio provides a shortcut for setting up a clickable view, and for associating an event handler with the view: use the `android:onClick` attribute with the clickable view's element in the XML layout.

For example, the following XML expression in the layout file for a Button sets `showToast()` as the event handler:

```
android:onClick="showToast"
```

When the `b`utton is tapped, its `android:onClick` attribute calls the `showToast()` method.

Write the event handler, such as `showToast()` referenced in the XML code above, to call other methods that implement the app's *model* logic:

```
public void showToast(View view) {
        // Do something in response to the button click.
}
```

In order to work with the `android:onClick` attribute, the `showToast()` method must be `public`, return `void`, and require a `view` parameter in order to know which view called the method.

Android Studio provides a shortcut for creating an event handler *stub* (a placeholder for the method that you can fill in later) in the Java code for the activity associated with the XML layout. Follow these steps:

1. Inside the XML layout file (such as activity_main.xml), click the method name in the `android:onClick` attribute statement.
2. Press Alt-Enter in Windows or Option-Return in Mac OS X, and select **Create onClick event handler**.
3. Choose the activity associated with the layout file (such as **MainActivity**) and click **OK**. This creates a placeholder

method stub in MainActivity.java.

Updating views

To update a view's contents, such as replacing the text in a TextView, your code must first instantiate an object from the view. Your code can then update the object, thereby updating the view.

To refer to the view in your code, use the findViewById() method of the View class, which looks for a view based on the resource `id`. For example, the following statement sets `mShowCount` to be the TextView with the resource id `show_count`:

```
mShowCount = (TextView) findViewById(R.id.show_count);
```

From this point on, your code can use `mShowCount` to represent the TextView, so that when you update `mShowCount`, the view is updated.

For example, when the following button with the `android:onClick` attribute is tapped, onClick calls the `countUp()` method:

```
android:onClick="countUp"
```

You can implement `countUp()` to increment the count, convert the count to a string, and set the string as the text for the `mShowCount` object:

```
public void countUp(View view) {
        mCount++;
        if (mShowCount != null)
            mShowCount.setText(Integer.toString(mCount));
}
```

Since you had already associated `mShowCount` with the TextView for displaying the count, the `mShowCount.setText()` method updates the text view on the screen.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Make Your First Interactive UI
- Using Layouts

Learn more

- Android Studio documentation:
  - Android Studio User Guide
- Android API Guide, "Develop" section:
  - UI Overview
  - Common Layout Objects
  - Color class definition
  - Android standard R.color resources
  - Supporting Different Densities
- Material Design:
  - Style
  - Using the Material Theme
- Other:
  - Android Studio User's Guide: Image Asset Studio
  - Model-View-Presenter (MVP) architecture pattern

- Hierarchy Viewer for visualizing the view hierarchy
- Color Hex Color Codes

1.3: Text and Scrolling Views

Contents:

- TextView
- Scrolling views
- Related practical
- Learn more

This chapter describes one of the most often used views in apps: the TextView, which shows textual content on the screen. A TextView can be used to show a message, a response from a database, or even entire magazine-style articles that users can scroll. This chapter also shows how you can create a scrolling view of text and other elements.

TextView

One view you may use often is the TextView class, which is a subclass of the View class that displays text on the screen. You can use TextView for a view of any size, from a single character or word to a full screen of text. You can add a resource `id` to the TextView, and control how the text appears using attributes in the XML layout file.

You can refer to a TextView view in your Java code by using its resource `id`, and update the text from your code. If you want to allow users to edit the text, use EditText, a subclass of TextView that allows text input and editing. You learn all about EditText in another chapter.

TextView attributes

You can use XML attributes to control:

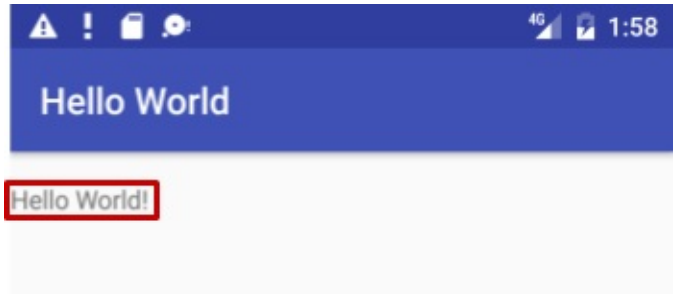
- Where the TextView is positioned in a layout (like any other view)
- How the view itself appears, such as with a background color
- What the text looks like within the view, such as the initial text and its style, size, and color

For example, to set the width, height, and position within a LinearLayout:

```
<TextView
   ...
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   … />
```

To set the initial text value of the view, use the `android:text` attribute:

```
android:text="Hello World!"
```



You can extract the text string into a string resource (perhaps called `hello_world`) that's easier to maintain for multiple-language versions of the app, or if you need to change the string in the future. After extracting the string, use the string resource name with `@string/` to specify the text:

```
android:text="@string/hello_world"
```

The most often used attributes with TextView are the following:

- android:text: Set the text to display.
- android:textColor: Set the color of the text. You can set the attribute to a color value, a predefined resource, or a theme. Color resources and themes are described in other chapters.
- android:textAppearance: The appearance of the text, including its color, typeface, style, and size. You set this attribute to a predefined style resource or theme that already defines these values.
- android:textSize: Set the text size (if not already set by `android:textAppearance`). Use `sp` (scaled-pixel) sizes such as `20sp` or `14.5sp`, or set the attribute to a predefined resource or theme.
- android:textStyle: Set the text style (if not already set by `android:textAppearance`). Use `normal`, `bold`, `italic`, or `bold|italic`.
- Android:typeface: Set the text typeface (if not already set by `android:textAppearance`). Use `normal`, `sans`, `serif`, or `monospace`.
- android:lineSpacingExtra: Set extra spacing between lines of text. Use `sp` (scaled-pixel) or dp (device-independent pixel) sizes, or set the attribute to a predefined resource or theme.
- android:autoLink: Controls whether links such as URLs and email addresses are automatically found and converted to clickable (touchable) links. Use one of the following:
  - `none`: Match no patterns (default).
  - `web`: Match web URLs.
  - `email`: Match email addresses.
  - `phone`: Match phone numbers.
  - `map`: Match map addresses.
  - `all`: Match all patterns (equivalent to web|email|phone|map).

  For example, to set the attribute to match web URLs, use this:

  ```
  android:autoLink="web"
  ```

Using embedded tags in text

In an app that accesses magazine or newspaper articles, the articles that appear would probably come from an online source or might be saved in advance in a database on the device. You can also create text as a single long string in the strings.xml resource.

In either case, the text may contain embedded HTML tags or other text formatting codes. To properly display in a text view, text must be formatted following these rules:

- If you have an apostrophe (') in your text, you must *escape* it by preceding it with a backslash (**\'**). If you have a double-quote in your text, you must also escape it (**\"**). You must also escape any other non-ASCII characters. See the "Formatting and Styling" section of String Resources for more details.
- The TextView ignores all HTML tags except the following:
  - Use the HTML and **** tags around words that should be in bold.
  - Use the HTML and **</i>** tags around words that should be in italics. Note, however, that if you use curled apostrophes within an italic phrase, you should replace them with straight apostrophes.
  - You can combine bold and italics by combining the tags, as in **...** words...**</i>**.

To create a long string of text in the **strings.xml** file, enclose the entire text within `<string name="your_string_name"></string>` in the strings.xml file (*your_string_name* is the name you provide the string resource, such as `article_text`).

Text lines in the strings.xml file don't wrap around to the next line — they extend beyond the right margin. This is the correct behavior. Each new line of text starting at the left margin represents an entire paragraph.

Enter **\n** to represent the end of a line, and another **\n** to represent a blank line. If you don't add end-of-line characters, the paragraphs will run into each other when displayed on the screen.

Tip: If you want to see the text wrapped in strings.xml, you can press Return to enter hard line endings, or format the text first in a text editor with hard line endings. The endings will not be displayed on the screen.

Referring to a TextView in code

To refer to a TextView in your Java code, use its resource `id`. For example, to update a TextView with new text, you would:

1. Find the TextView (with the id `show_count`) and assign it to a variable. You use the `findViewById()` method of the View class, and refer to the view you want to find using this format:

   ```
   R.id.view_id
   ```

   In which `view_id` is the resource identifier for the view:

   ```
   mShowCount = (TextView) findViewById(R.id.show_count);
   ```

2. After retrieving the view as a TextView member variable, you can then set the text of the text view to the new text using the setText() method of the TextView class:

   ```
   mShowCount.setText(mCount_text);
   ```

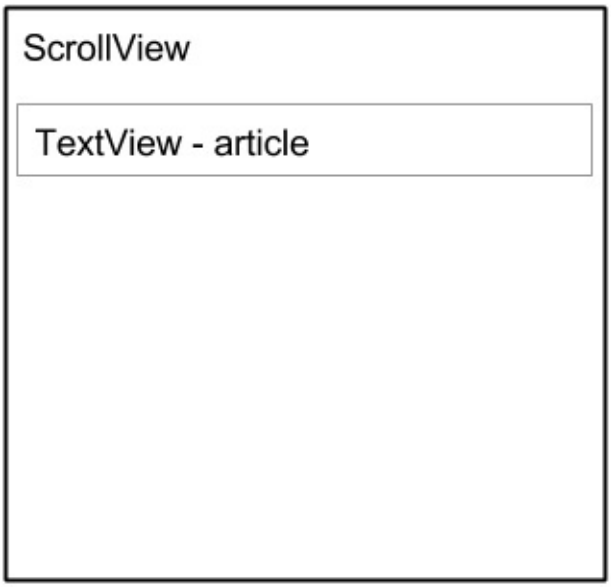
Scrolling views

If the information you want to show in your app is larger than the device's display, you can create a *scrolling view* that the user can scroll vertically by swiping up or down, or horizontally by swiping right or left.

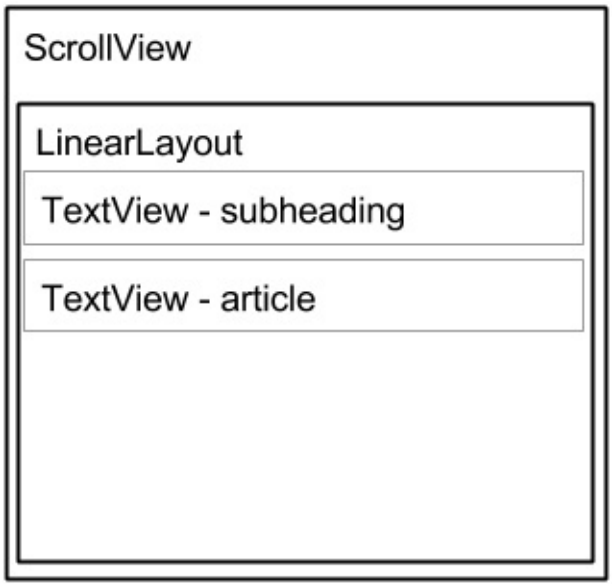
You would typically use a scrolling view for news stories, articles, or any lengthy text that doesn't completely fit on the display. You can also use a scrolling view to combine views (such as a TextView and a Button) within a scrolling view.

Creating a layout with a ScrollView

The ScrollView class provides the layout for a vertical scrolling view. (For horizontal scrolling, you would use HorizontalScrollView.) ScrollView is a subclass of FrameLayout, which means that you can place only *one* view as a child within it; that child contains the entire contents to scroll.



Even though you can place only one child view inside a ScrollView, the child view could be a view group with a hierarchy of child views, such as a LinearLayout. A good choice for a view within a ScrollView is a LinearLayout that is arranged in a vertical orientation.



ScrollView and performance

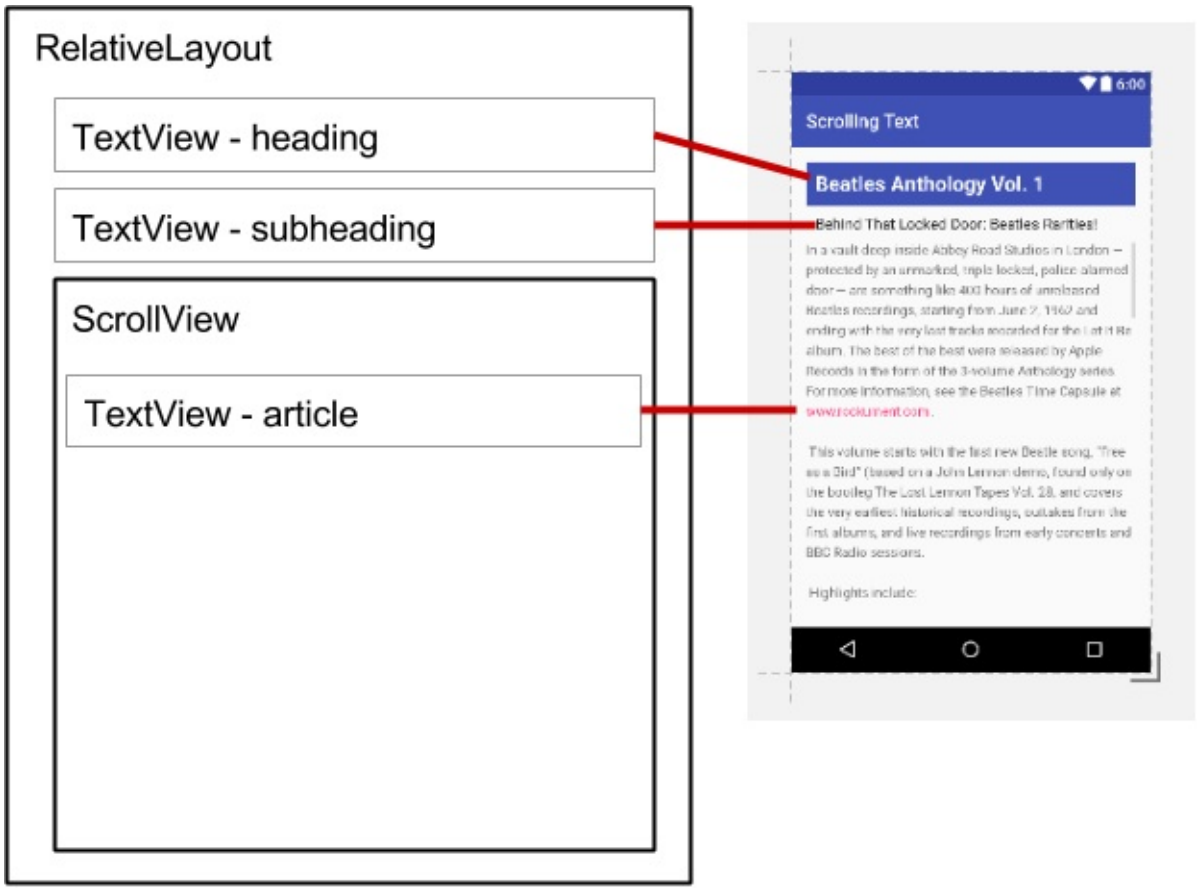
With a ScrollView, all of your views are in memory and in the view hierarchy even if they aren't displayed on screen. This makes ScrollView useful for smoothly scrolling pages of free-form text, because the text is already in memory. However, ScrollView can use up a lot of memory, which can affect the performance of the rest of your app.

Using nested instances of LinearLayout can also lead to an excessively deep view hierarchy, which can slow down performance. Nesting several instances of LinearLayout that use the `android:layout_weight` attribute can be especially expensive as each child view needs to be measured twice. Consider using flatter layouts such as RelativeLayout or GridLayout to improve performance.

Complex layouts with ScrollView may suffer performance issues, especially with child views such as images. We recommend that you *not* use images with a ScrollView. To display long lists of items, or images, consider using a RecyclerView. Also, using AsyncTask provides a simple way to perform work outside the main thread, such as loading images in a background thread, then applying them to the UI once finished. AsyncTask is covered in another chapter.

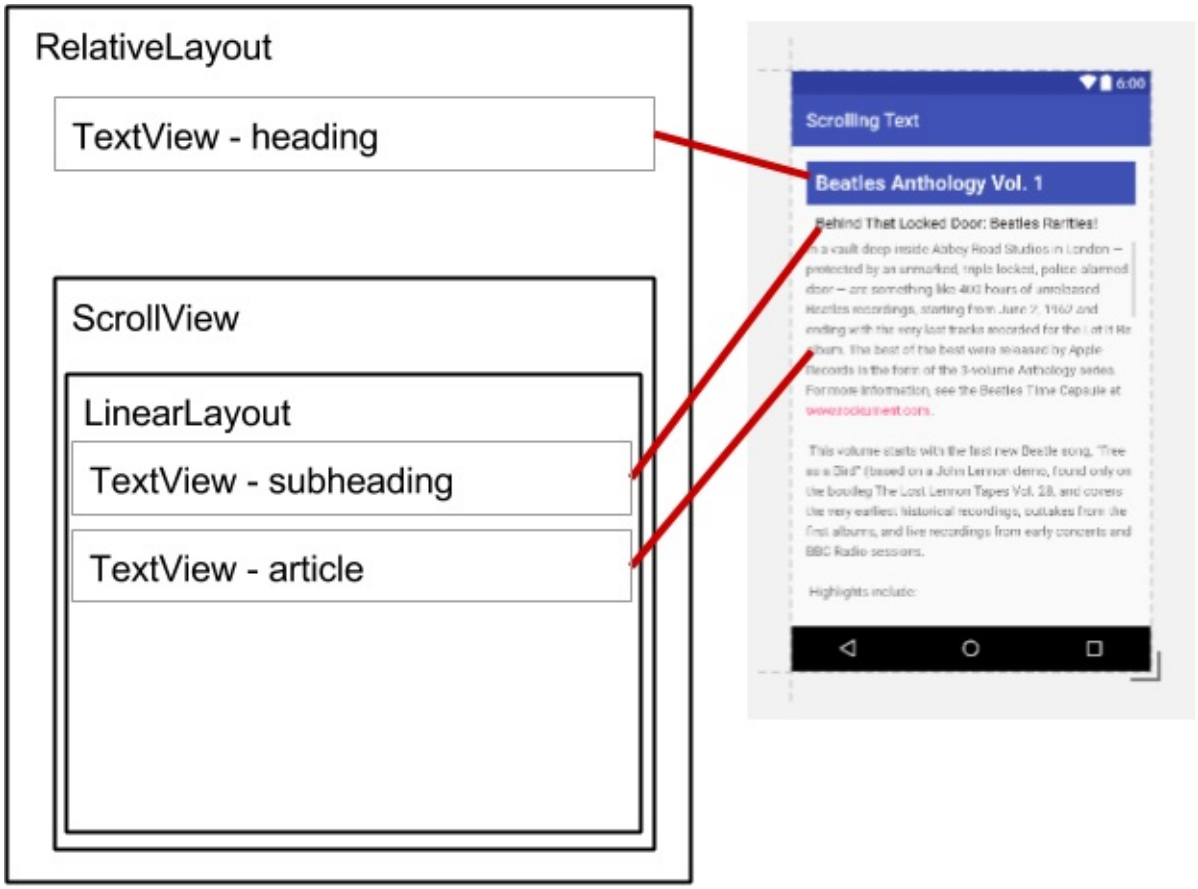
ScrollView with a TextView

To show a scrollable magazine article on the screen, you might use a RelativeLayout for the screen that includes a separate TextView for the article heading, another for the article subheading, and a third TextView for the scrolling article text (see figure below), set within a ScrollView. The only part of the screen that would scroll would be the ScrollView with the article text.



ScrollView with a LinearLayout

The ScrollView view group can contain only one view; however, that view can be a view group that contains views, such as LinearLayout. You can *nest* a view group such as LinearLayout *within* the ScrollView view group, thereby scrolling everything that is inside the LinearLayout.



When adding a LinearLayout inside a ScrollView, use `match_parent` for the LinearLayout's `android:layout_width` attribute to match the width of the parent view group (the ScrollView), and use `wrap_content` for the LinearLayout's `android:layout_height` attribute to make the view group only big enough to enclose its contents and padding.

Since ScrollView only supports vertical scrolling, you must set the LinearLayout orientation to vertical (by using the `android:orientation="vertical"` attribute), so that the entire LinearLayout will scroll vertically. For example, the following XML layout scrolls the `article` TextView along with the `article_subheading` TextView:

```
<ScrollView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/article_heading">

   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <TextView
         android:id="@+id/article_subheading"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:padding="@dimen/padding_regular"
         android:text="@string/article_subtitle"
         android:textAppearance="@android:style/TextAppearance" />

      <TextView
         android:id="@+id/article"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:autoLink="web"
         android:lineSpacingExtra="@dimen/line_spacing"
         android:text="@string/article_text" />

   </LinearLayout>
</ScrollView>
```

```
<ScrollView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_below="@id/article_heading">

   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">

      <TextView
         android:id="@+id/article_subheading"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:padding="@dimen/padding_regular"
         android:text="@string/article_subtitle"
         android:textAppearance="@android:style/TextAppearance" />

      <TextView
         android:id="@+id/article"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:autoLink="web"
         android:lineSpacingExtra="@dimen/line_spacing"
         android:text="@string/article_text" />

   </LinearLayout>
</ScrollView>
```

Related practicals

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Working with TextView Elements

Learn more

- Android Studio documentation:
  - Meet Android Studio
  - Android Studio User Guide
- Android API Guide, "Develop" section:
  - TextView
  - ScrollView
  - String Resources
  - View
  - Relative Layout
- Material Design:
  - Style
  - Using the Material Theme
- Other:
  - Android Developers Blog: Linkify your Text!
  - Codepath: Working with a TextView

1.4: Resources to Help You Learn

Contents:

- Exploring Android developer documentation
- Watching developer videos
- Exploring code samples in the Android SDK
- Using activity templates
- Browsing the Android developer blog
- Other sources of information
- Related practical

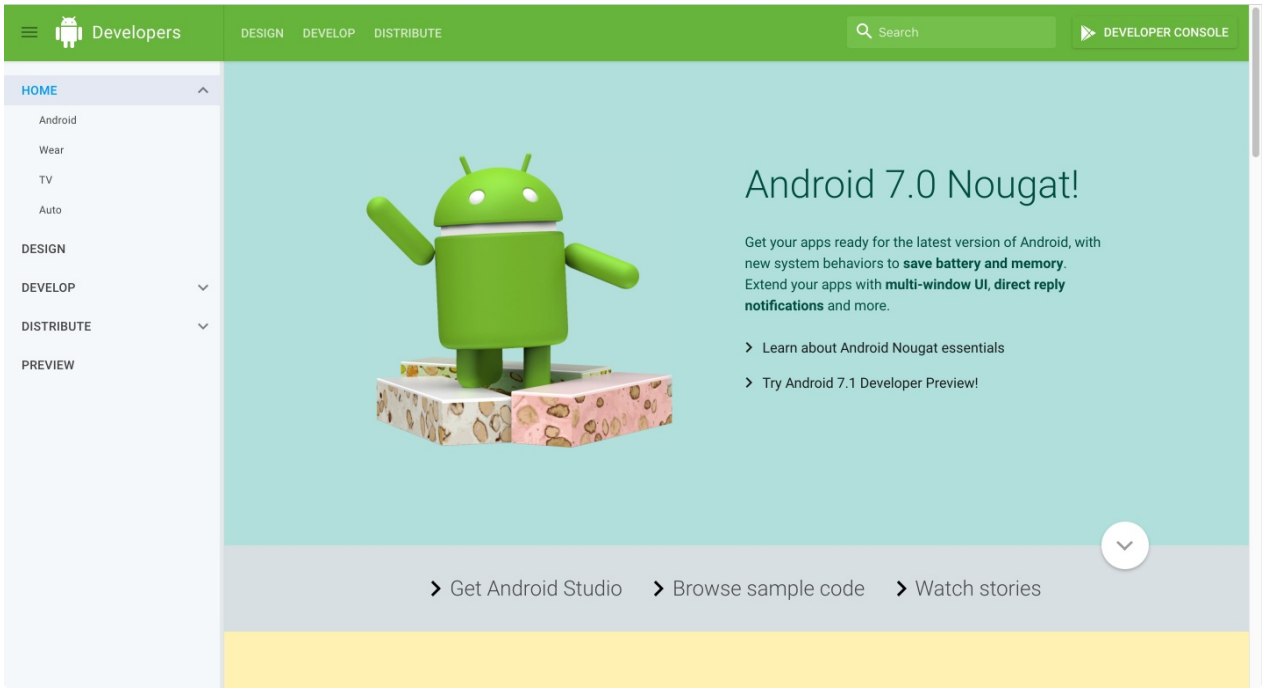
This chapter describes resources available for Android developers, and how to use them.

Exploring Android developer documentation

The best place to learn about Android development and to keep informed about the newest Android development tools is to browse the official Android developer documentation.

developer.android.com

Home page



This documentation contains a wealth of information kept current by Google. To start exploring, click the following links on the home page:

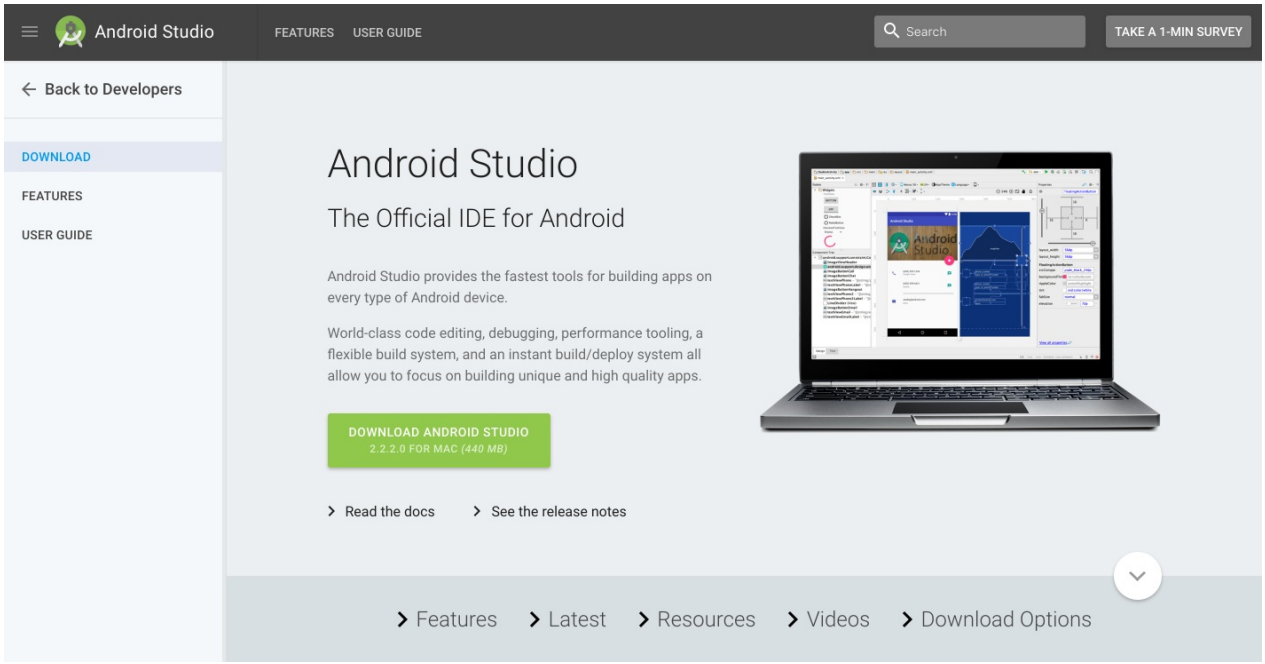
- **Get Android Studio**: Download Android Studio, the official integrated development environment (IDE) for building Android apps.
- **Browse sample code**: Browse the sample code library in GitHub to learn how to build different components for your apps. Click the categories in the left column to browse the available samples. Each sample is a fully functioning Android app. You can browse the resources and source files, and see the overall project structure. Copy and paste the code you need, and if you want to share a link to a specific line you can double-click it to the get the URL. For more sample code, see "Exploring code samples in the Android SDK" in this chapter.

- **Watch stories**: Learn about other Android developers, their apps, and their successes with Android and Google Play. The page offers videos and articles with the newest stories about Android development, such as how developers improved their users experiences, and how to increase user engagement with apps.

The home page also offers links for Android developers to preview their apps for the newest version of Android, and to join the Google Play developer program:

- **Developer Console**: The Google Play store is Google's digital distribution system for apps developed with the Android SDK. On the Google Play Developer Console page you can accept the Developer Agreement, pay the registration fee, and complete your account details in order to join the Google Play developer program.
- **Preview**: Go to the preview page for the newest version of Android to test your apps for compatibility, and to take advantage of new features like app shortcuts, image keyboard support, circular icons, and more.
- **Android**, **Wear**, **TV**, and **Auto**: Learn about the newest versions of Android for smartphones and tablets, wearable devices, television, and automobiles.

Android Studio page


Android Studio
FEATURES
USER GUIDE
Search
TAKE A 1-MIN SURVEY
Back to Developers
DOWNLOAD
FEATURES
USER GUIDE
Android Studio
The Official IDE for Android
Android Studio provides the fastest tools for building apps on every type of Android device.
World-class code editing, debugging, performance tooling, a flexible build system, and an instant build/deploy system all allow you to focus on building unique and high quality apps.
DOWNLOAD ANDROID STUDIO
2.2.2.0 FOR MAC (440 MB)
Read the docs
See the release notes
Features
Latest
Resources
Videos
Download Options

After clicking **Get Android Studio** on the home page, the Android Studio page, shown above, appears with the following useful links:

- **Download Android Studio**: Download Android Studio for the computer operating system you are currently using.
- **Read the docs**: Browse the Android Studio documentation.
- **See the release notes**: Read the release notes for the newest version of Android Studio.
- **Features**: Learn about the features of the newest version of Android Studio.
- **Latest**: Read news about Android Studio.
- **Resources**: Read articles about using Android Studio, including a basic introduction.
- **Videos**: Watch video tutorials about using Android Studio.
- **Download Options**: Download a version of Android Studio for a different operating system than the one you are using.

Android Studio documentation

The following are links into the Android Studio documentation that are useful for this training:

- Meet Android Studio
- Developer Workflow Basics
- Projects Overview
- Create App Icons with Image Asset Studio

- Add Multi-Density Vector Graphics
- Create and Manage Virtual Devices
- Android Monitor page
- Debug Your App
- Configure Your Build
- Sign Your App

Design, Develop, Distribute, and Preview

The Android documentation is accessible through the following links from the home page:

- **Design**: This section covers Material Design, which is a conceptual design philosophy that outlines how apps should look and work on mobile devices. Use the following links to learn more:
  - **Introducing material design**: An introduction to the material design philosophy.
  - **Downloads for designers**: Download color palettes for compatibility with the material design specification.
  - **Articles**: Read articles and news about Android design.
  - Scroll down the Design page for links to resources such as videos, templates, font, and color palettes.
  - The following are links into the Design section that are useful for this training:
    - Material Design Guidelines
    - Style
    - Using the Material Theme
    - Components - Buttons
    - Dialogs design guide
    - Gestures design guide
    - Notification Design Guide
    - Icons and other downloadable resources
    - Design - Patterns - Navigation
    - Drawable Resource Guide
    - Styles and Themes Guide
    - Settings
    - Material Palette Generator
- **Develop**: This section is where you can find application programming interface (API) information, reference documentation, tutorials, tool guides, and code samples, and gain insights into Android's tools and libraries to speed your development. You can use the site navigation links in the left column, or search to find what you need. The following are popular links into the Develop section that are useful for this training:
  - Overview:
    - Introduction to Android
    - Vocabulary Glossary
    - Platform Architecture
    - Android Application Fundamentals
    - UI Overview
    - Platform Versions
    - Android Support Library
    - Working with System Permissions
  - Development practices:
    - Supporting Different Platform Versions
    - Supporting Multiple Screens
    - Supporting Different Densities
    - Best Practices for Interaction and Engagement
    - Best Practices for User Interface
    - Best Practices for Testing
    - Providing Resources
    - Optimizing Downloads for Efficient Network Access Guide
    - Best Practices for App Permissions

    - Articles and training guides:
        - Starting Another Activity
        - Specifying the Input Method Type
        - Handling Keyboard Input
        - Adding the App Bar
        - Using Touch Gestures
        - Creating Lists and Cards
        - Getting Started with Testing
        - Managing the Activity Lifecycle
        - Connecting to the Network
        - Managing Network Usage
        - Manipulating Broadcast Receivers On Demand
        - Scheduling Repeating Alarms
        - Transferring Data Without Draining the Battery
        - Saving Files
        - Saving Key-Value Sets
        - Saving Data in SQL Databases
        - Configuring Auto Backup for Apps
        - Working with System Permissions
    - General topics
        - Styles and Themes
        - Layouts
        - Menus
        - Intents and Intent Filters
        - Processes and threads
        - Loaders
        - Services
        - Notifications
        - Storage Options
        - Localizing with Resources
        - Content Providers
        - Cursors
        - Backing up App Data to the Cloud
        - Settings
        - System Permissions
    - Reference information:
        - Support Library
        - Android standard R.color resources
        - App Resources
- **Distribute**: This section provides information about everything that happens after you've written your app: putting it on the Play Store, growing your user base, and earning money.
    - Google Play
    - Essentials for a Successful App
    - Launch Checklist

Installing offline documentation

To access to documentation even when you are not connected to the internet, install the Software Development Kit (SDK) documentation using the SDK Manager. Follow these steps:

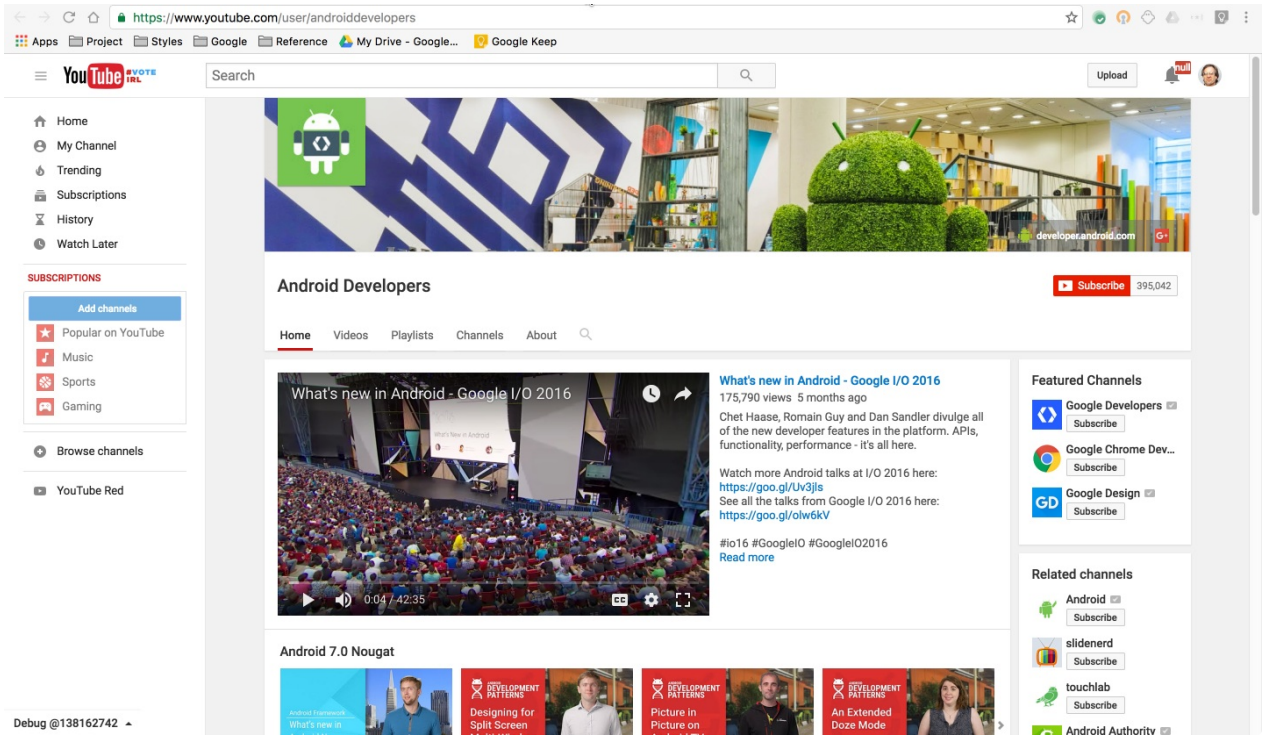
1. Choose **Tools > Android > SDK Manager**.
2. In the left column, click **Android SDK**.
3. Select and copy the path for the Android SDK Location at the top of the screen, as you will need it to locate the documentation on your computer:

Appearance & Behavior › System Settings › Android SDK
Manager for the Android SDK and Tools used by Android Studio
Android SDK Location: /Users/ /Library/Android/sdk
Edit
SDK Platforms
SDK Tools
SDK Update Sites

4. Click the **SDK Tools** tab. You can install additional SDK Tools that are not installed by default, as well as an offline version of the Android developer documentation.
5. Click the checkbox for "Documentation for Android SDK" if it is not already installed, and click **Apply**.
6. When the installation finishes, click **Finish**.
7. Navigate to the **sdk** directory you copied above, and open the **docs** directory.
8. Find **index.html** and open it.

Watching developer videos

In addition to the Android documentation, the Android Developer YouTube channel is a great source of tutorials and tips. You can subscribe to the channel to receive notifications of new videos by email. To subscribe, click the red **Subscribe** button in the upper right corner as shown below.



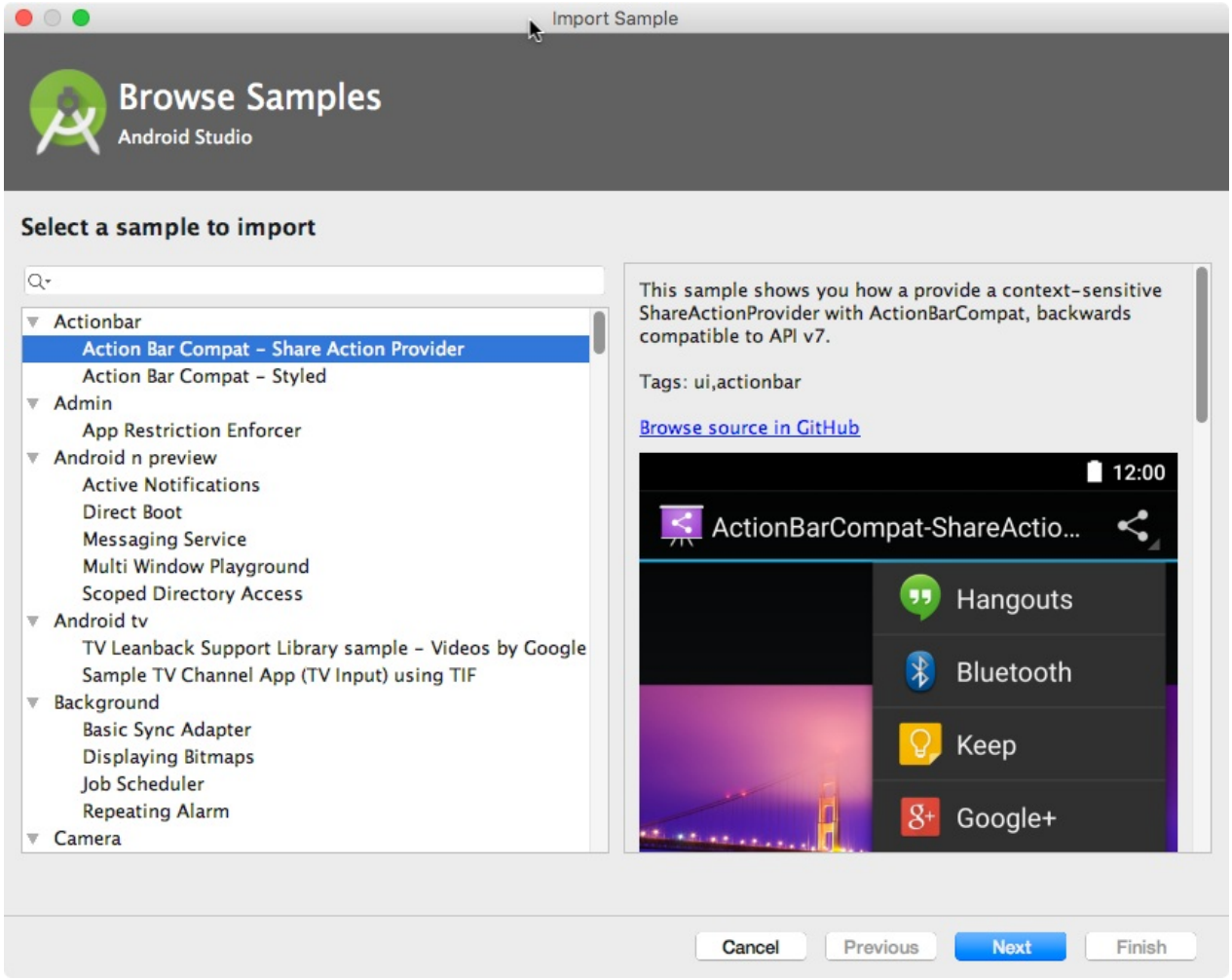
The following are popular videos referred to in this training:

- Debugging and Testing in Android Studio
- Android Testing Support - Android Testing Patterns #1
- Android Testing Support - Android Testing Patterns #2
- Android Testing Support - Android Testing Patterns #3
- Threading Performance 101
- Good AsyncTask Hunting
- Scheduling Alarms Presentation
- RecyclerView Animations and Behind the Scenes (Android Dev Summit 2015)
- Android Application Architecture: The Next Billion Users
- Android Performance Patterns Playlist

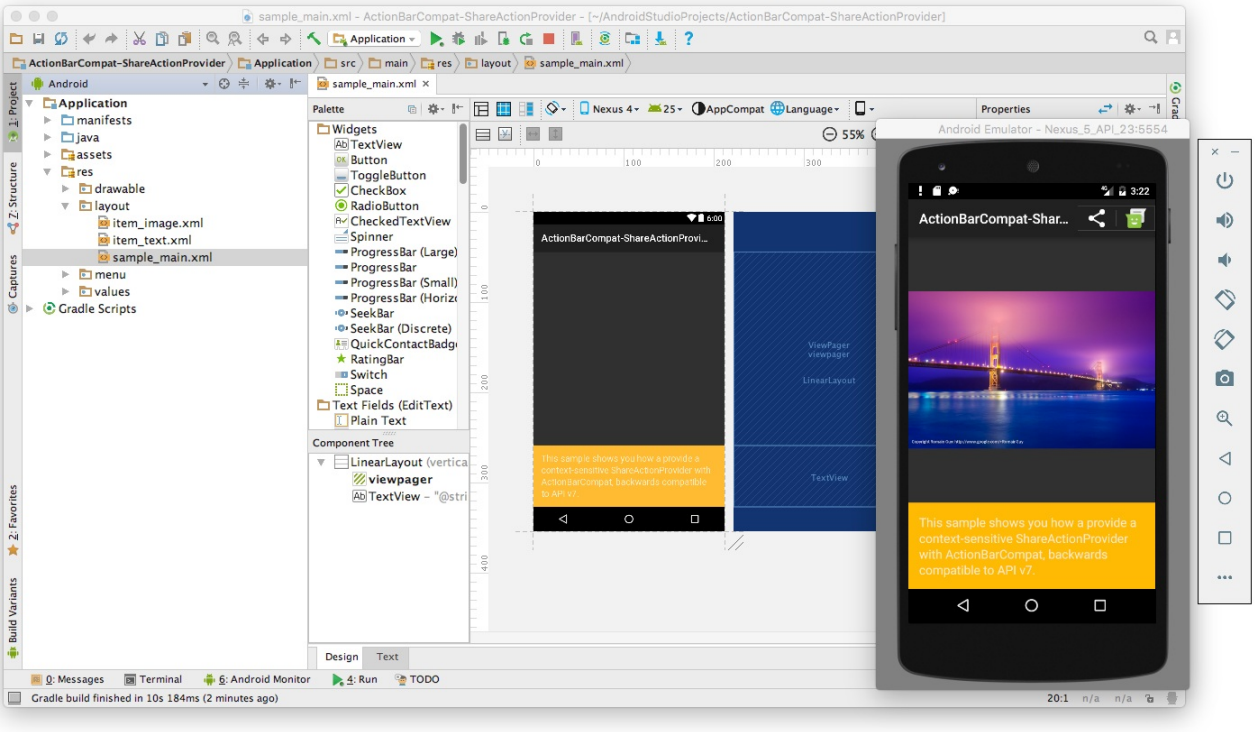
In addition, Udacity offers online Android development courses.

Exploring code samples in the Android SDK

You can explore hundreds of code samples directly in Android Studio. Choose **Import an Android code sample** from the Android Studio welcome screen, or choose **File > New > Import Sample** if you have already opened a project. The Browse Samples window appears as shown below.



Choose a sample and click **Next**. Accept or edit the Application name and Project location, and click **Finish**. The app project appears as shown below, and you can run the app in the emulator provided with Android Studio, or on a connected device.

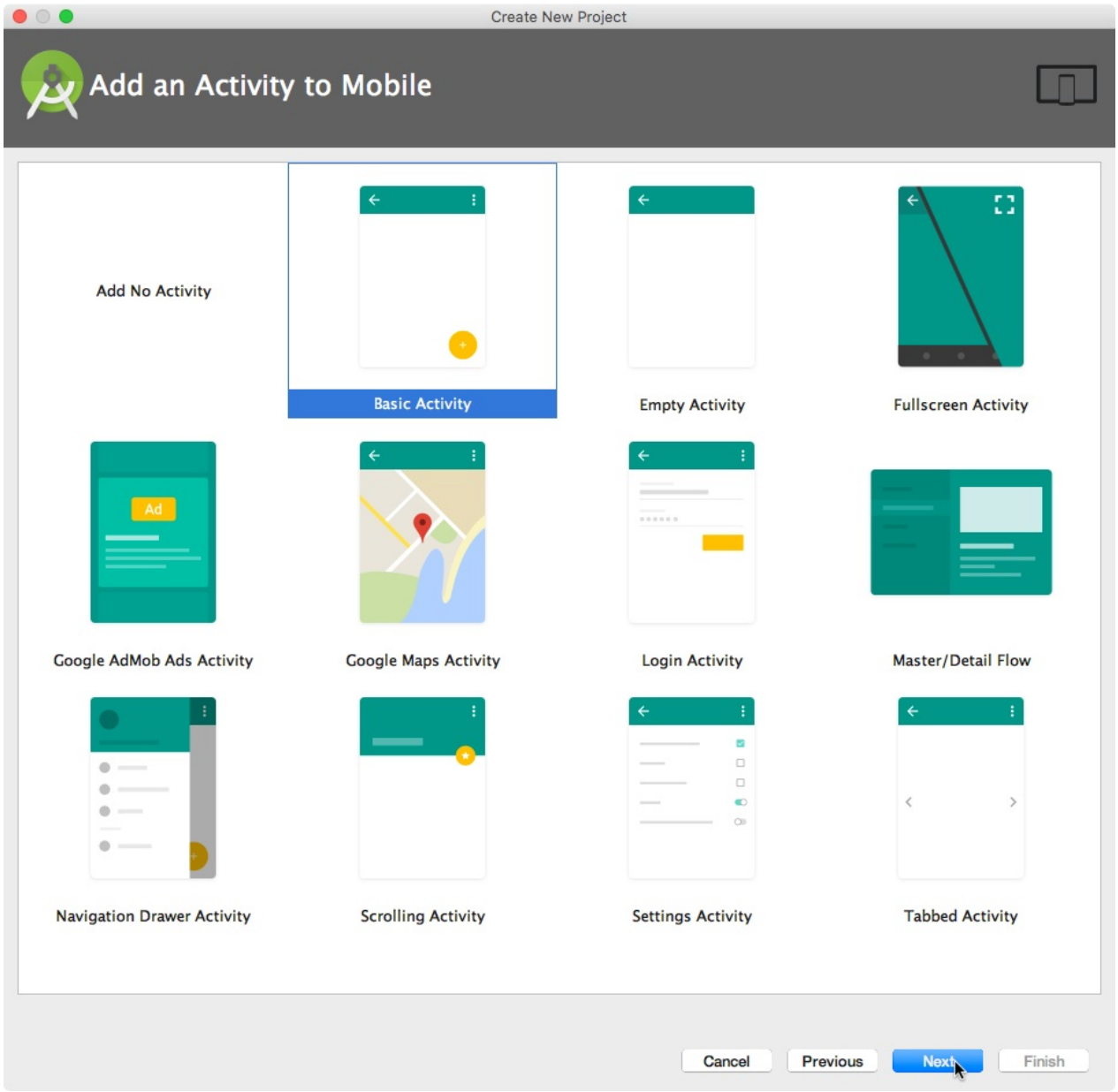


Note: The samples are meant to be a starting point for further development. We encourage you to design and build your own ideas into them.

Using activity templates

Android Studio provides templates for common and recommended activity designs. Using templates saves time, and helps you follow best practices for developing activities.

Each template incorporates an skeleton activity and user interface. You choose an activity template for the main activity when starting an app project. You can also add an activity template to an existing project. Right-click the **java** folder in the Project: Android view and choose **New > Activity > Gallery**.



Browsing the Android developer blog

The Android Developers Blog provides a wealth of articles on Android development.

The following are popular blog posts:

- Android Studio 2.2
- Keeping Android safe: Security enhancements in Nougat
- Connecting your App to a Wi-Fi Device
- Linkify your Text!
- Holo Everywhere
- Tips to help you stay on the right side of Google Play policy
- 5 Tips to help you improve game-as-a-service monetization

Other sources of information

Google and third parties offer a wide variety of helpful tips and techniques for Android development. The following are sources of information referenced by this training:

- **Google Developer Training**: Whether you're new to programming or an experienced developer, Google offers a range of online courses to teach you Android development, from getting started to optimizing app performance. Click the **Android** tab at the top of the page.
- **Google I/O Codelabs**: Google Developers Codelabs provide a guided hands-on coding experience on a number of topics. Most codelabs will step you through the process of building a small app, or adding a new feature to an existing app. Choose **Android** from the Category drop-down menu on the right side of the page.
- **Android Testing Codelab**: This codelab shows you how to get started with testing for Android, including testing integration in Android Studio, unit testing, hermetic testing, functional user interface testing, and the Espresso testing framework.
- **Google Testing Blog**: This blog is focused on testing code. Blog posts referred to in the training include:
  - Android UI Automated Testing
  - Test Sizes
- **Stack Overflow**: Stack Overflow is a community of millions of programmers helping each other. If you run into a problem, chances are someone else has already posted an answer on this forum. Examples referred to in the training include:
  - How to assert inside a RecyclerView in Espresso?
  - How do I Add A Fragment to a Custom Navigation Drawer Template?
  - How do you create Preference Activity and Preference Fragment on Android?
  - How to use SharedPreferences in Android to store, fetch and edit values
  - How to populate AlertDialog from Arraylist?
  - onSavedInstanceState vs. SharedPreferences
  - Glide vs. Picasso
- **Google on GitHub**: GitHub is a Git repository hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. Git is a widely used version control system for software development. The following are hosted within GitHub and referred to in this training:
  - Android Testing Samples
  - Android Testing Support Library: Espresso basics
  - Android Testing Support Library: Espresso cheat sheet
  - Roman Nurik's Android Asset Studio
  - Source code for exercises on GitHub
- Miscellaneous sources of information referred to in this training:
  - Codepath: Working with a TextView
  - SQLite.org: Full description of the Query Language
  - Atomic Object: "Espresso – Testing RecyclerViews at Specific Positions"
- **Google search**: Enter a question into the Google search box, prefaced by "Android" to narrow your search. The Google search engine will collect relevant results from all of these resources. For example:
  - *"What is the most popular Android OS version in India?"* This question collects results about Android market share, including the Dashboards page that provides an overview of device characteristics and platform versions that are active in the Android ecosystem.
  - *"Android Settings Activity"* collects various articles about the Settings Activity including the Settings topic page, the PreferenceActivity class, and Stack Overflow's How do you create Preference Activity and Preference Fragment on Android?
  - *"Android TextView"* collects information about text views including the TextView class, the View class, the Layouts topic page, and code samples from various sources.
  - Preface any search with *"Android"* to narrow your search to Android-related topics. For example, you can search for any Android class description, such as *"Android TextView"* or *"Android activity"*.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Learning About Available Resources

2.1: Understanding Activities and Intents

Contents:

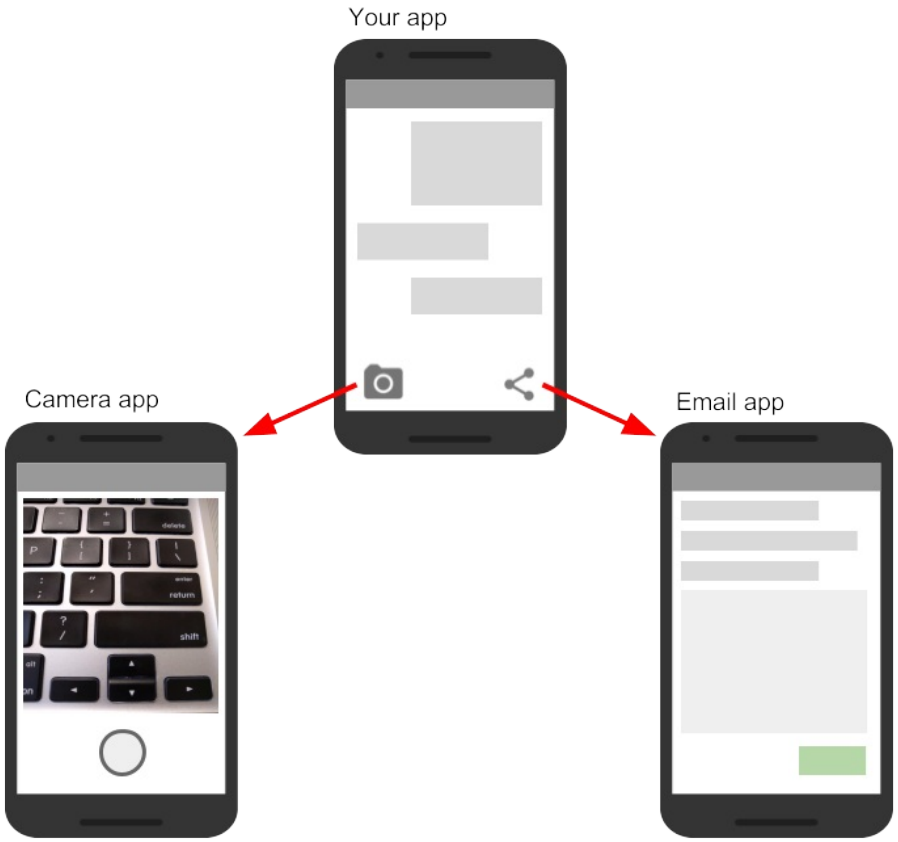
- Introduction
- About activities
- Creating activities
- About intents
- Starting an activity with an explicit intent
- Passing data between activities with intents
- Getting data back from an activity
- Activity navigation
- Related practical
- Learn more

In this chapter you'll learn about *activities*, the major building blocks of your app's user interface, as well as using *intents* to communicate between activities.

About activities

An activity represents a single screen in your app with an interface the user can interact with. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading individual messages. Your app is a collection of activities that you either create yourself, or that you reuse from other apps.

Although the activities in your app work together to form a cohesive user experience in your app, each one is independent of the others. This enables your app to start activities in other apps, and other apps can start your activities (if your app allows it). For example, a messaging app you write could start an activity in a camera app to take a picture, and then start the activity in an email app to let the user share that picture in email.



Typically, one activity in an app is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start other activities in order to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes.

When an activity is stopped because a new activity starts, the first activity is notified of that change with the activity's lifecycle callback methods. The Activity lifecycle is the set of states an activity can be in, from when it is first created, to each time it is stopped or resumed, to when the system destroys it. You'll learn more about the activity lifecycle in the next chapter.

Creating activities

To implement an activity in your app, do the following:

- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.

When you create a new project for your app, or add a new activity to your app, in Android Studio (with File > New > Activity), template code for each of these tasks is provided for you.

Create the activity class

Activities are subclasses of the Activity class, or one of its subclasses. When you create a new project in Android Studio, your activities are, by default, subclasses of the AppCompatActivity class. The AppCompatActivity class is a subclass of Activity that lets you to use up-to-date Android app features such as the action bar and material design, while still enabling

your app to be compatible with devices running older versions of Android.

Here is a skeleton subclass of AppCompatActivity:

```
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
   }
}
```

The first task for you in your activity subclass is to implement the standard activity lifecycle callback methods (such as OnCreate()) to handle the state changes for your activity. These state changes include things such as when the activity is created, stopped, resumed, or destroyed. You'll learn more about the activity lifecycle and lifecycle callbacks in the next chapter.

The one required callback your app must implement is the onCreate() method. The system calls this method when it creates your activity, and all the essential components of your activity should be initialized here. Most importantly, the OnCreate() method calls setContentView() to create the primary layout for the activity.

You typically define the user interface for your activity in one or more XML layout files. When the setContentView() method is called with the path to a layout file, the system creates all the initial views from the specified layout and adds them to your activity. This is often referred to as *inflating* the layout.

You may often also want to implement the onPause() method in your activity class. The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back). You'll learn more about onPause() and all the other lifecycle callbacks in the next chapter.

In addition to lifecycle callbacks, you may also implement methods in your activity to handle other behavior such as user input or button clicks.

Implement a user interface

the user interface for an activity is provided by a hierarchy of views, which controls a particular space within the activity's window and can respond to user interaction.

The most common way to define a user interface using views is with an XML layout file stored as part of your app's resources. Defining your layout in XML enables you to maintain the design of your user interface separately from the source code that defines the activity's behavior.

You can also create new views directly in your activity code by inserting new view objects into a ViewGroup, and then passing the root ViewGroup to setContentView(). After your layout has been inflated -- regardless of its source -- you can add more views in Java anywhere in the view hierarchy.

Declare the activity in the manifest

Each activity in your app must be declared in the Android app manifest with the `<activity>` element, inside `<application>`. When you create a new project or add a new activity to your project in Android Studio, your manifest is created or updated to include skeleton activity declarations for each activity. Here's the declaration for the main activity.

```
<activity android:name=".MainActivity" >
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
</activity>
```

The `<activity>` element includes a number of attributes to define properties of the activity such as its label, icon, or theme. The only required attribute is android:name, which specifies the class name for the activity (such as "MainActivity"). See the `<activity>` element reference for more information on activity declarations.

The `<activity>` element can also include declarations for intent filters. The intent filters specify the kind of intents your activity will accept.

```
 <intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

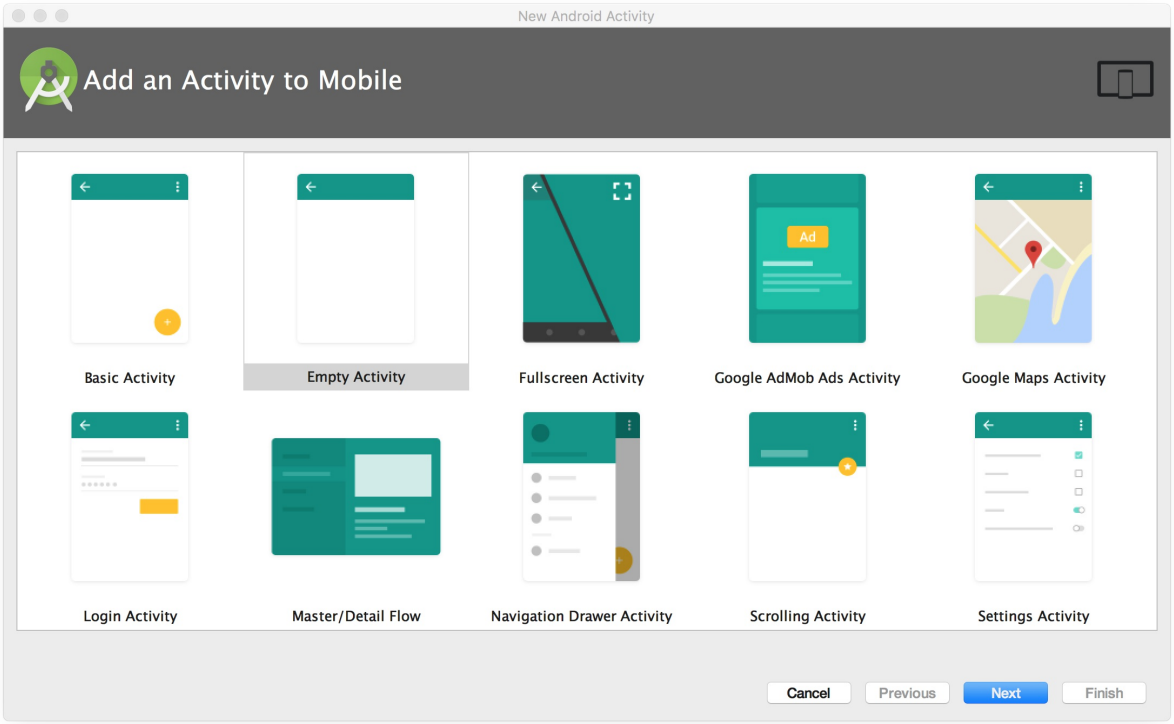
Intent filters must include at least one `<action>` element, and can also include a `<category>` and optional `<data>`. The main activity for your app needs an intent filter that defines the "main" action and the "launcher" category so that the system can launch your app. Android Studio creates this intent filter for the main activity in your project:

The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

Other activities in your app can also declare intent filters, but only your main activity should include the "main" action.. You'll learn more about implicit intents and intent filters in a later section.

Add more activities to your project

The main activity for your app and its associated layout file comes with your project when you create it. You can add new activities to your project in Android Studio with the **File > New > Activity** menu. Choose the activity template you want to use, or open the Gallery to see all the available templates.



When you choose an activity template, you'll see the same set of screens for creating the new activity that you did when you initially created the project. Android Studio provides these three things for each new activity in your app:

- A Java file for the new activity with a skeleton class definition and onCreate() method. The new activity, like the main activity, is a subclass of AppCompatActivity.
- An XML file containing the layout for the new activity. Note that the setContentView() method in the activity class

inflates this new layout.
- An additional `<activity>` element in the Android manifest that specifies the new activity. The second activity definition does not include any intent filters. If you intend to use this activity only within your app (and not enable that activity to be started by any other app), you do not need to add filters.

About intents

All Android activities are started or activated with an *intent*. Intents are message objects that make a request to the Android runtime to start an activity or other app component in your app or in some other app. You don't start those activities yourself;

When your app is first started from the device home screen, the Android runtime sends an intent to your app to start your app's main activity (the one defined with the MAIN action and the LAUNCHER category in the Android Manifest). To start other activities in your app, or request that actions be performed by some other activity available on the device, you build your own intents with the Intent class and call the startActivity() method to send that intent.

In addition to starting activities, intents are also used to pass data between activities. When you create an intent to start a new activity, you can include information about the data you want that new activity to operate on. So, for example, an email activity that displays a list of messages can send an intent to the activity that displays that message. The display activity needs data about the message to display, and you can include that data in the intent.

In this chapter you'll learn about using intents with activities, but intents are also used to start services and broadcast receivers. You'll learn about both those app components later on in the book.

Intent types

There are two types of intents in Android:

- *Explicit intents* specify the receiving activity (or other component) by that activity's fully-qualified class name. Use an explicit intent to start a component in your own app (for example, to move between screens in the user interface), because you already know the package and class name of that component.
- *Implicit intents* do not specify a specific activity or other component to receive the intent. Instead you declare a general action to perform in the intent. The Android system matches your request to an activity or other component that can handle your requested action. You'll learn more about implicit intents in a later chapter.

Intent objects and fields

An Intent object is an instance of the Intent class. For explicit intents, the key fields of an intent include the following:

- The activity *class* (for explicit intents). This is the class name of the activity or other component that should receive the intent, for example, com.example.SampleActivity.class. Use the intent constructor or the intent's setComponent(), setComponentName() or setClassName() methods to specify the class.
- The intent *data*. The intent data field contains a reference to the data you want the receiving activity to operate on, as a Uri object.
- Intent *extras*. These are key-value pairs that carry information the receiving activity requires to accomplish the requested action.
- Intent *flags*. These are additional bits of metadata, defined by the Intent class. The flags may instruct the Android system how to launch an activity or how to treat it after it's launched.

For implicit intents, you may need to also define the intent action and category. You'll learn more about intent actions and categories in section 2.3.

Starting an activity with an explicit intent

To start a specific activity from another activity, use an explicit intent and the startActivity() method. Explicit intents include the fully-qualified class name for the activity or other component in the Intent object. All the other intent fields are optional, and null by default.

For example, if you wanted to start the ShowMessageActivity to show a specific message in an email app, use code like this.

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
startActivity(messageIntent);
```

The Intent constructor takes two arguments for an explicit intent.

- An application context. In this example, the activity class provides the content (here, `this`).
- The specific component to start (`ShowMessageActivity.class`).

Use the startActivity() method with the new intent object as the only argument. The startActivity() method sends the intent to the Android system, which launches the ShowMessageActivity class on behalf of your app. The new activity appears on the screen, and the originating activity is paused.

The started activity remains on the screen until the user taps the back button on the device, at which time that activity closes and is reclaimed by the system, and the originating activity is resumed. You can also manually close the started activity in response to a user action (such as a button click) with the finish() method:

```
public void closeActivity (View view) {
    finish();
}
```

Passing data between activities with intents

In addition to simply starting one activity from another, you also use intents to pass information between activities. The intent object you use to start an activity can include intent *data* (the URI of an object to act on), or intent *extras*, which are bits of additional data the activity might need.

In the first (sending) activity, you:

1. Create the Intent object.
2. Put data or extras into that intent.
3. Start the new activity with startActivity().

In the second (receiving) activity, you:

1. Get the intent object the activity was started with.
2. Retrieve the data or extras from the Intent object.

When to use intent data or intent extras

You can use either intent data and intent extras to pass data between the activities. There are several key differences between data and extras that determine which you should use.

The intent *data* can hold only one piece of information. A URI representing the location of the data you want to operate on. That URI could be a web page URL (http://), a telephone number (tel://), a goegraphic location (geo://) or any other custom URI you define.

Use the intent data field:

- When you only have one piece of information you need to send to the started activity.
- When that information is a data location that can be represented by a URI.

Intent *extras* are for any other arbitrary data you want to pass to the started activity. Intent extras are stored in a Bundle object as key and value pairs. Bundles are a map, optimized for Android, where the keys are strings, and the values can be any primitive or object type (objects must implement the Parcelable interface). To put data into the intent extras you can use any of the Intent class's putExtra() methods, or create your own bundle and put it into the intent with putExtras().

Use the intent extras:

- If you want to pass more than one piece of information to the started activity.
- If any of the information you want to pass is not expressible by a URI.

Intent data and extras are not exclusive; you can use data for a URI and extras for any additional information the started activity needs to process the data in that URI.

Add data to the intent

To add data to an explicit intent from the originating activity, create the intent object as you did before:

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use the setData() method with a Uri object to add that URI to the intent. Some examples of using setData() with URIs:

```
// A web page URL
messageIntent.setData(Uri.parse("http://www.google.com"));
// a Sample file URI
messageIntent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
// A sample content: URI for your app's data model
messageIntent.setData(Uri.parse("content://mysample.provider/data"));
// Custom URI
messageIntent.setData(Uri.parse("custom:" + dataID + buttonId));
```

Keep in mind that the data field can only contain a single URI; if you call setData() multiple times only the last value is used. Use intent extras to include additional information (including URIs.)

After you've added the data, you can start the activity with the intent as usual.

```
startActivity(messageIntent);
```

Add extras to the intent

To add intent extras to an explicit intent from the originating activity:

1. Determine the keys to use for the information you want to put into the extras, or define your own. Each piece of information needs its own unique key.
2. Use the putExtra() methods to add your key/value pairs to the intent extras. Optionally you can create a Bundle object, add your data to the bundle, and then add the bundle to the intent.

The Intent class includes several intent extra keys you can use, defined as constants that begin with the word EXTRA_. For example, you could use Intent.EXTRA_EMAIL to indicate an array of email addresses (as strings), or Intent.EXTRA_REFERRER to specify information about the originating activity that sent the intent.

You can also define your own intent extra keys. Conventionally you define intent extra keys as static variables with names that begin with EXTRA_. To guarantee that the key is unique, the string value for the key itself should be prefixed with your app's fully qualified class name. For example:

```
public final static String EXTRA_MESSAGE = "com.example.mysampleapp.MESSAGE";
public final static String EXTRA_POSITION_X = "com.example.mysampleapp.X";
public final static String EXTRA_POSITION_Y = "com.example.mysampleapp.Y";
```

Create an intent object (if one does not already exist):

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
```

Use a putExtra() method with a key to put data into the intent extras. The Intent class defines many putExtra() methods for different kinds of data:

```
messageIntent.putExtra(EXTRA_MESSAGE, "this is my message");
messageIntent.putExtra(EXTRA_POSITION_X, 100);
messageIntent.putExtra(EXTRA_POSITION_Y, 500);
```

Alternately, you can create a new bundle and populate that bundle with your intent extras. Bundle defines many "put" methods for different kinds of primitive data as well as objects that implement Android's Parcelable interface or Java's Serializable.

```
Bundle extras = new Bundle();
extras.putString(EXTRA_MESSAGE, "this is my message");
extras.putInt(EXTRA_POSITION_X, 100);
extras.putInt(EXTRA_POSITION_Y, 500);
```

After you've populated the bundle, add it to the intent with the putExtras() method (note the "s" in Extras):

```
messageIntent.putExtras(extras);
```

Start the activity with the intent as usual:

```
startActivity(messageIntent);
```

Retrieve the data from the intent in the started activity

When you start an activity with an intent, the started activity has access to the intent and the data it contains.

To retrieve the intent the activity (or other component) was started with, use the getIntent() method:

```
Intent intent = getIntent();
```

Use getData() to get the URI from that intent:

```
Uri locationUri = getData();
```

To get the extras out of the intent, you'll need to know the keys for the key/value pairs. You can use the standard Intent extras if you used those, or you can use the keys you defined in the originating activity (if they were defined as public.)

Use one of the getExtra() methods to extract extra data out of the intent object:

```
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
int positionX = intent.getIntExtra(MainActivity.EXTRA_POSITION_X);
int positionY = intent.getIntExtra(MainActivity.EXTRA_POSITION_Y);
```

Or you can get the entire extras bundle from the intent and extract the values with the various Bundle methods:

```
Bundle extras = intent.getExtras();
String message = extras.getString(MainActivity.EXTRA_MESSAGE);
```

Getting data back from an activity

When you start an activity with an intent, the originating activity is paused, and the new activity remains on the screen until the user clicks the back button, or you call the finish() method in a click handler or other function that ends the user's involvement with this activity.

Sometimes when you send data to an activity with an intent, you would like to also get data back from that intent. For example, you might start a photo gallery activity that lets the user pick a photo. In this case your original activity needs to receive information about the photo the user chose back from the launched activity.

To launch a new activity and get a result back, do the following steps in your originating activity:

1. Instead of launching the activity with startActivity(), call startActivityForResult() with the intent and a request code.
2. Create a new intent in the launched activity and add the return data to that intent.
3. Implement onActivityResult() in the originating activity to process the returned data.

You'll learn about each of these steps in the following sections.

Use startActivityForResult() to launch the activity

To get data back from a launched activity, start that activity with the startActivityForResult() method instead of startActivity().

```
startActivityForResult(messageIntent, TEXT_REQUEST);
```

The startActivityForResult() method, like startActivity(), takes an intent argument that contains information about the activity to be launched and any data to send to that activity. The startActivityForResult() method, however, also needs a request code.

The request code is an integer that identifies the request and can be used to differentiate between results when you process the return data. For example, if you launch one activity to take a photo and another to pick a photo from a gallery, you'll need different request codes to identify which request the returned data belongs to.

Conventionally you define request codes as static integer variables with names that include REQUEST. Use a different integer for each code. For example:

```
public static final int PHOTO_REQUEST = 1;
public static final int PHOTO_PICK_REQUEST = 2;
public static final int TEXT_REQUEST = 3;
```

Return a response from the launched activity

The response data from the launched activity back to the originating activity is sent in an intent, either in the data or the extras. You construct this return intent and put the data into it in much the same way you do for the sending intent. Typically your launched activity will have an onClick or other user input callback method in which you process the user's action and close the activity. This is also where you construct the response.

To return data from the launched activity, create a new empty intent object.

```
Intent returnIntent = new Intent();
```

Note: To avoid confusing sent data with returned data, use a new intent object rather than reusing the original sending intent object.
Return result intents do not need a class or component name to end up in the right place. The Android system directs the response back to the originating activity for you.

Add data or extras to the intent the same way you did with the original intent. You may need to define keys for the return intent extras at the start of your class.

```
public final static String EXTRA_RETURN_MESSAGE =
    "com.example.mysampleapp.RETURN_MESSAGE";
```

Then put your return data into the intent as usual. Here the return message is an intent extra with the key EXTRA_RETURN_MESSAGE.

```
messageIntent.putExtra(EXTRA_RETURN_MESSAGE, mMessage);
```

Use the setResult() method with a response code and the intent with the response data:

```
setResult(RESULT_OK,replyIntent);
```

The response codes are defined by the Activity class, and can be

- RESULT_OK. the request was successful.
- RESULT_CANCELED: the user cancelled the operation.
- RESULT_FIRST_USER. for defining your own result codes.

You'll use the result code in the originating activity.

Finally, call finish() to close the activity and resume the originating activity:

```
finish();
```

Read response data in onActivityResult()

Now that the launched activity has sent data back to the originating activity with an intent, that first activity must handle that data. To handle returned data in the originating activity, implement the onActivityResult() callback method. Here is a simple example.

```
public void onActivityResult(int requestCode, int resultCode,  Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == TEXT_REQUEST) {
        if (resultCode == RESULT_OK) {
            String reply =
                data.getStringExtra(SecondActivity.EXTRA_RETURN_MESSAGE);
                // process data
        }
    }
}
```

The three arguments to the onActivityResult() contain all the information you need to handle the return data.

- **Request code.** The request code you set when you launched the activity with startActivityForResult(). If you launch different activities to accomplish different operations, use this code to identify the specific data you're getting back.
- **Result code:** the result code set in the launched activity, usually one of RESULT_OK or RESULT_CANCELED.
- **Intent data**. the intent that contains the data returned from the launch activity.

The example method shown above shows the typical logic for handling the request and response codes. The first test is for the TEXT_REQUEST request, and that the result was successful. Inside the body of those tests you extract the return information out of the intent. Use getData() to get the intent data, or getExtra() to retrieve values out of the intent extras with a specific key.

Activity navigation

Any app of any complexity that you build will include multiple activities, both designed and implemented by you, and potentially in other apps as well. As your users move around your app and between activities, consistent navigation becomes more important to the app's user experience. Few things frustrate users more than basic navigation that behaves in inconsistent and unexpected ways. Thoughtfully designing your app's navigation will make using your app predictable and reliable for your users.

Android system supports two different forms of navigation strategies for your app.

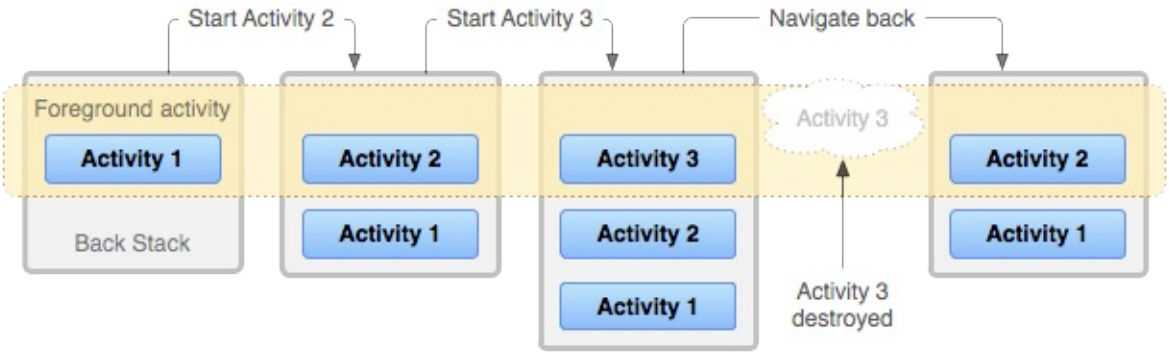
- Temporal or Back navigation, provided by the device back button, and the back stack.
- Ancestral, or Up navigation, provided by you as an option in the app's action bar.

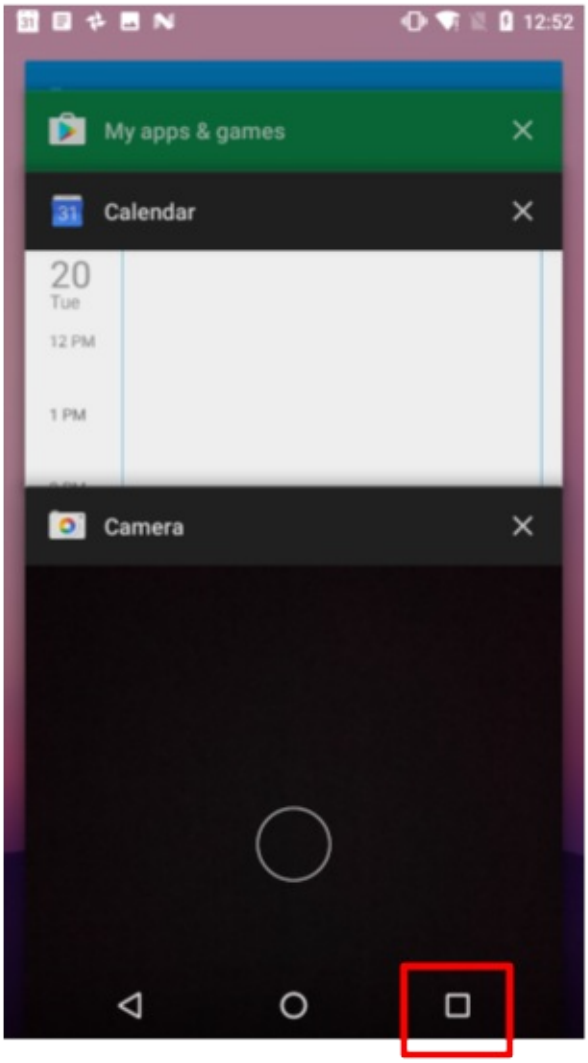
Back navigation, tasks, and the back stack

Back navigation allows your users to return to the previous activity by tapping the device back button . Back navigation is also called *temporal* navigation because the back button navigates the history of recently viewed screens, in reverse chronological order.

The *back stack* is the set of activities that the user has visited and that can be returned to by the user with the back button. Each time a new activity starts, it is pushed onto the back stack and takes user focus. The previous activity is stopped but is still available in the back stack. The back stack operates on a "last in, first out" mechanism, so when the user is done with the current activity and presses the Back button, that activity is popped from the stack (and destroyed) and the previous activity resumes.

Because an app can start activities both inside and outside a single app, the back stack contains all the activities that have been launched by the user in reverse order. Each time the user presses the Back button, each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen.





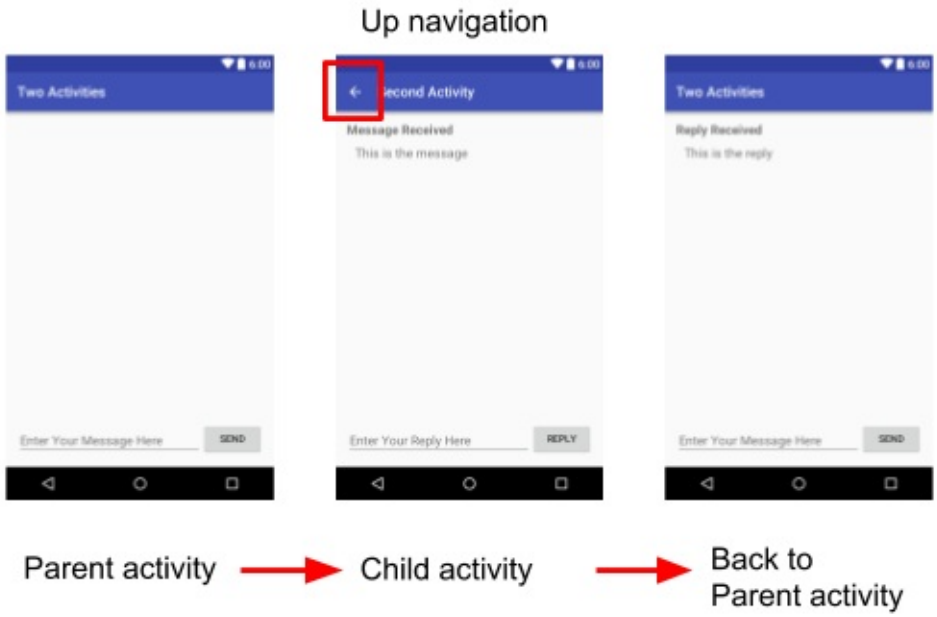
In most cases you don't have to worry about managing either tasks or the back stack for your app—the system keeps track of these things for you, and the back button is always available on the device.

There may, however, be times where you may want to override the default behavior for tasks or for the back stack. For example, if your screen contains an embedded web browser where users can navigate between web pages, you may wish to use the browser's default back behavior when users press the device's *Back* button, rather than returning to the previous activity. You may also need to change the default behavior for your app in other special cases such as with notifications or widgets, where activities deep within your app may be launched as their own tasks, with no back stack at all. You'll learn more about managing tasks and the back stack in a later section.

Up navigation

Up navigation, sometimes referred to as ancestral or logical navigation, is used to navigate within an app based on the explicit hierarchical relationships between screens. With Up navigation, your activities are arranged in a hierarchy, and "child" activities show a left-facing arrow in the action bar that returns the user to the "parent" activity. The topmost activity in the hierarchy is usually your main activity, and the user cannot go up from there.

Up navigation, sometimes referred to as ancestral or logical navigation, is used to navigate within an app based on the explicit hierarchical relationships between screens. With Up navigation, your activities are arranged in a hierarchy, and "child" activities show a left-facing arrow in the action bar that returns the user to the "parent" activity. The topmost activity in the hierarchy is usually your main activity, and the user cannot go up from there.



For instance, if the main activity in an email app is a list of all messages, selecting a message launches a second activity to display that single email. In this case the message activity would provide an Up button that returns to the list of messages.

The behavior of the Up button is defined by you in each activity based on how you design your app's navigation. In many cases, Up and Back navigation may provide the same behavior: to just return to the previous activity. For example, a Settings activity may be available from any activity in your app, so "up" is the same as back -- just return the user to their previous place in the hierarchy.

Providing Up behavior for your app is optional, but a good design practice, to provide consistent navigation for the activities in your app.

Implement up navigation with parent activities

With the standard template projects in Android Studio, it's straightforward to implement Up navigation. If one activity is a child of another activity in your app's activity hierarchy, specify that activity's parent in the Android Manifest.

Beginning in Android 4.1 (API level 16), declare the logical parent of each activity by specifying the android:parentActivityName attribute in the `<activity>` element. To support older versions of Android, include `<meta-data>` information to define the parent activity explicitly. Use both methods to be backwards-compatible with all versions of Android.

Here are the skeleton definitions for both a main (parent) activity and a second (child) activity:

```
<application ... >
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name=".MainActivity" ...>
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

    </activity>
    <!-- A child of the main activity -->
    <activity android:name=".SecondActivity"
        android:label="@string/activity2_name"
        android:parentActivityName=".MainActivity">
        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value="com.example.android.twoactivities.MainActivity" />
    </activity>
</application>
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Create and Start Activities

Learn more

- Android Application Fundamentals
- Starting Another Activity
- Activity (API Guide)
- Activity (API Reference)
- Intents and Intent Filters (API Guide)
- Intent (API Reference)

2.2: The Activity Lifecycle and Managing State

Contents:

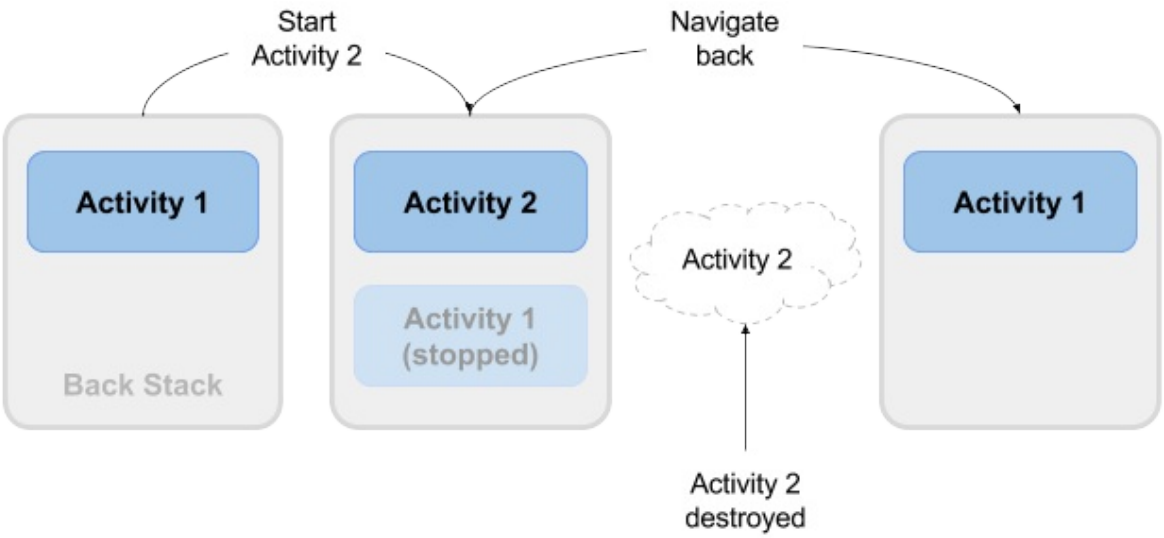
- Introduction
- About the activity lifecycle
- Activity states and lifecycle callback methods
- Configuration changes and activity state
- Related Practical
- Learn More

In this chapter you'll learn about the activity lifecycle, the callback events you can implement to perform tasks in each stage of the lifecycle, and how to handle activity instance states throughout the activity lifecycle.

About the activity lifecycle

The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it is initially created to when it is destroyed and the system reclaims that activity's resources. As the user interacts with your app and other apps on the device, the different activities move into different states.

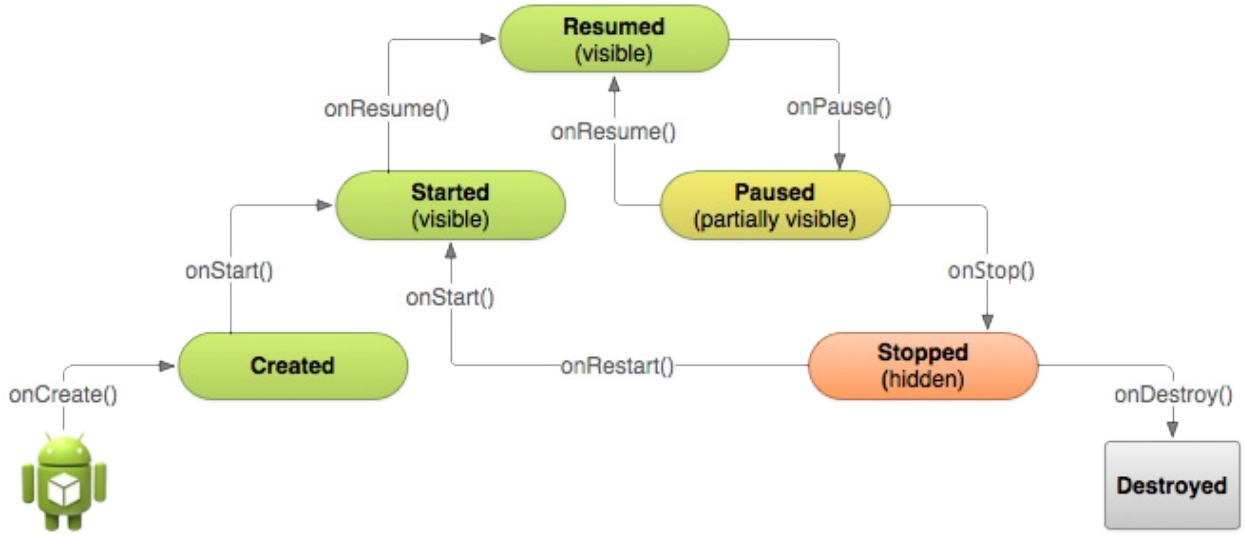
For example, when you start an app, the app's main activity (Activity 1) is started, comes to the foreground, and receives the user focus. When you start a second activity (Activity 2), that new activity is also created and started, and the main activity is stopped. When you're done with the second activity and navigate back, the first activity resumes. The second activity stops and is no longer needed; if the user does not resume the second activity, it is eventually destroyed by the system.



Activity states and lifecycle callback methods

When an activity transitions into and out of the different lifecycle states as it runs, the Android system calls several lifecycle callback methods at each stage. All of the callback methods are hooks that you can override in each of your Activity classes to define how that activity behaves when the user leaves and re-enters the activity. Keep in mind that the lifecycle states (and callbacks) are per activity, not per app, and you may implement different behavior at different points in the lifecycle for different activities in your app.

This figure shows each of the activity states and the callback methods that occur as the activity transitions between different states:



Depending on the complexity of your activity, you probably don't need to implement all the lifecycle callback methods in your activities. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application.

Activity created (onCreate() method)

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // The activity is being created.
}
```

Your activity enters into the created state when it is started for the first time. When an activity is first created the system calls the onCreate() method to initialize that activity. For example, when the user taps your app icon from the Home screen to start that app, the system calls the onCreate() method for the activity in your app that you've declared to be the "launcher" or "main" activity. In this case the main activity's onCreate() method is analogous to the main() method in other programs.

Similarly, if your app starts another activity with an intent (either explicit or implicit), the system matches your intent request with an activity and calls onCreate() for that new activity.

The onCreate() method is the only required callback you must implement in your activity class. In your onCreate() method you perform basic application startup logic that should happen only once, such as setting up the user interface, assigning class-scope variables, or setting up background tasks.

Created is a transient state; the activity remains in the created state only as long as it takes to run onCreate(), and then the activity moves to the started state.

Activity started (onStart() method)

```
@Override
protected void onStart() {
    super.onStart();
    // The activity is about to become visible.
}
```

After your activity is initialized with onCreate(), the system calls the onStart() method, and the activity is in the started state. The onStart() method is also called if a stopped activity returns to the foreground, such as when the user clicks the back or up buttons to navigate to the previous screen. While onCreate() is called only once when the activity is created, the onStart() method may be called many times during the lifecycle of the activity as the user navigates around your app.

When an activity is in the started state and visible on the screen, the user cannot interact with it until onResume() is called, the activity is running, and the activity is in the foreground.

Typically you implement onStart() in your activity as a counterpart to the onStop() method. For example, if you release hardware resources (such as GPS or sensors) when the activity is stopped, you can re-register those resources in the onStart() method.

Started, like created, is a transient state. After starting the activity moves into the resumed (running) state.

Activity resumed/running (onResume() method)

```
@Override
protected void onResume() {
    super.onResume();
    // The activity has become visible (it is now "resumed").
}
```

Your activity is in the resumed state when it is initialized, visible on screen, and ready to use. The resumed state is often called the running state, because it is in this state that the user is actually interacting with your app.

The first time the activity is started the system calls the onResume() method just after onStart(). The onResume() method may also be called multiple times, each time the app comes back from the paused state.

As with the onStart() and onStop() methods, which are implemented in pairs, you typically only implement onResume() as a counterpart to onPause(). For example, if in the onPause() method you halt any onscreen animations, you would start those animations again in onResume().

The activity remains in the resumed state as long as the activity is in the foreground and the user is interacting with it. From the resumed state the activity can move into the paused state.

Activity paused (onPause() method)

```
@Override
protected void onPause() {
    super.onPause();
    // Another activity is taking focus
    // (this activity is about to be "paused").
}
```

The paused state can occur in several situations:

- The activity is going into the background, but has not yet been fully stopped. This is the first indication that the user is leaving your activity.
- The activity is only partially visible on the screen, because a dialog or other transparent activity is overlaid on top of it.
- In multi-window or split screen mode (API 24), the activity is displayed on the screen, but some other activity has the user focus.

The system calls the onPause() method when the activity moves into the paused state. Because the onPause() method is the first indication you get that the user may be leaving the activity, you can use onPause() to stop animation or video playback, release any hardware-intensive resources, or commit unsaved activity changes (such as a draft email).

The onPause() method should execute quickly. Don't use onPause() for for CPU-intensive operations such as writing persistent data to a database. The app may still be visible on screen as it passed through the paused state, and any delays in executing onPause() can slow the user's transition to the next activity. Implement any heavy-load operations when the

app is in the stopped state instead.

Note that in multi-window mode (API 24), your paused activity may still fully visible on the screen. In this case you do not want to pause animations or video playback as you would for a partially visible activity. You can use the inMultiWindowMode() method in the Activity class to test whether your app is running in multiwindow mode.

Your activity can move from the paused state into the resumed state (if the user returns to the activity) or to the stopped state (if the user leaves the activity altogether).

Activity stopped (onStop() method)

```
@Override
protected void onStop() {
    super.onStop();
    // The activity is no longer visible (it is now "stopped")
}
```

An activity is in the stopped state when it is no longer visible on the screen at all. This is usually because the user has started another activity, or returned to the home screen. The system retains the activity instance in the back stack, and if the user returns to that activity it is restarted again. Stopped activities may be killed altogether by the Android system if resources are low.

The system calls the onStop() method when the activity stops. Implement the onStop() method to save any persistent data and release any remaining resources you did not already release in onPause(), including those operations that may have been too heavyweight for onPause().

Activity destroyed (onDestroy() method)

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // The activity is about to be destroyed.
}
```

When your activity is destroyed it is shut down completely, and the Activity instance is reclaimed by the system. This can happen in several cases:

- You call finish() in your activity to manually shut it down.
- The user navigates back to the previous activity.
- The device is in a low memory situation where the system reclaims stopped activities to free more resources.
- A device configuration change occurs. You'll learn more about configuration changes later in this chapter.

Use onDestroy() to fully clean up after your activity so that no component (such as a thread) is running after the activity is destroyed.

Note that there are situations where the system will simply kill the activity's hosting process without calling this method (or any others), so you should not rely on onDestroy() to save any required data or activity state. Use onPause() or onStop() instead.

Activity restarted (onRestart() method)

```
@Override
protected void onRestart() {
    super.onRestart();
    // The activity is about to be restarted.
}
```

The restarted state is a transient state that only occurs if a stopped activity is started again. In this case the onRestart() method is called in between onStop() and onStart(). If you have resources that need to be stopped or started you typically implement that behavior in onStop() or onStart() rather than onRestart().

Configuration changes and activity state

Earlier in the section onDestroy() you learned that your activities may be destroyed when the user navigates back, by you with the finish() method, or by the system when it needs to free resources. The fourth time your activities are destroyed is when the device undergoes a *configuration change*.

Configuration changes occur on the device, in runtime, and invalidate the current layout or other resources in your activity The most common form of a configuration change is when the device is rotated. When the device rotates from portrait to landscape, or vice versa, the layout for your app also needs to change. The system recreates the activity to help that activity adapt to the new configuration by loading alternative resources (such as a landscape-specific layout).

Other configuration changes can include a change in locale (the user chooses a different system language), or the user enters multi-window mode (Android 7). In multi-window mode, if you have configured your app to be resizeable, Android recreates your activities to use a layout definition for the new, smaller activity size.

When a configuration change occurs Android system shuts down your activity (calling onPause(), onStop(), and onDestroy()), and then starts it over again from the start (calling onCreate(), onStart(), and onResume()).

Activity instance state

When an activity is destroyed and recreated, there are implications for the runtime state of that activity. When an activity is paused or stopped, the state of the activity is retained because that activity is still held in memory. When an activity is recreated, the state of the activity and any user progress in that activity is lost, with these exceptions:

- Some activity state information is automatically saved by default. The state of views in your layout with a unique ID (as defined by the android:id attribute in the layout) are saved and restored when an activity is recreated. In this case, the user-entered values in EditText views are usually retained when the activity is recreated.
- The intent that was used to start the activity, and the information stored in that intent's data or extras, remains available to that activity when it is recreated.

The activity state is stored as a set of key/value pairs in a Bundle object called the *activity instance state*. The system saves default state information to instance state bundle just before the activity is stopped, and passes that bundle to the new activity instance to restore.

You can add your own instance data to the instance state bundle by overriding the onSaveInstanceState() callback. The state bundle is passed to the onCreate() method, so you can restore that instance state data when your activity is created. There is also a corresponding onRestoreInstanceState() callback you can use to restore the state data.

Because device rotation is a common use case for you app, make sure you test that your activity behaves correctly in response to this configuration change, and implement instance state if you need to.

Note: The activity instance state is particular to a specific instance of an activity, running in a single task. If the user force-quits the app, reboots the device, or if the Android system shuts down the entire app process to preserve memory, the activity instance state is lost. To keep state changes across app instances and device reboots, you need to write that data to shared preferences. You'll learn more about shared preferences in a later chapter.

Saving activity instance state

To save information to the instance state bundle, use the onSaveInstanceState() callback. This is not a lifecycle callback method, but it is called when the user is leaving your activity (sometime before the onStop() method).

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    // save your state data to the instance state bundle
}
```

The onSaveInstanceState() method is passed a Bundle object (a collection of key/value pairs) when it is called. This is the instance state bundle to which you will add your own activity state information.

You learned about bundles in a previous chapter when you added keys and values to the intent extras. Add information to the instance state bundle in the same way, with keys you define and the various "put" methods defined in the Bundle class:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);

    // Save the user's current game state
    savedInstanceState.putInt("score", mCurrentScore);
    savedInstanceState.putInt("level", mCurrentLevel);
}
```

Don't forget to call through to the superclass, to make sure the state of the view hierarchy is also saved to the bundle.

Restoring activity instance state

Once you've saved the activity instance state, you also need to restore it when the activity is recreated. You can do this one of two places:

- The onCreate() callback method, which is called with the instance state bundle when the activity is created.
- The onRestoreInstanceState() callback, which is called after onStart() after the activity is created.

Most of the time the better place to restore the activity state is in onCreate(), to ensure that your user interface including the state is available as soon as possible.

To restore the saved instances state in onCreate(), test for the existence of a state bundle before you try to get data out of it. When your activity is started for the first time there will be no state and the bundle will be null.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt("score");
        mCurrentLevel = savedInstanceState.getInt("level");
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Related Practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Activity Lifecycle and Instance State

Learn More

- Activities (API Guide)
- Activity (API Reference)
- Managing the Activity Lifecycle
- Pausing and Resuming an Activity
- Stopping and Restarting an Activity
- Recreating an Activity
- Handling Runtime Changes
- Bundle (API Reference)

2.3: Activities and Implicit Intents

Contents:

- Introduction
- About implicit intents
- Sending implicit intents
- Receiving implicit intents
- Sharing data with ShareCompat.IntentBuilder
- Managing tasks and activities
- Activity Launch Modes
- Task Affinities
- Related Practical
- Learn More

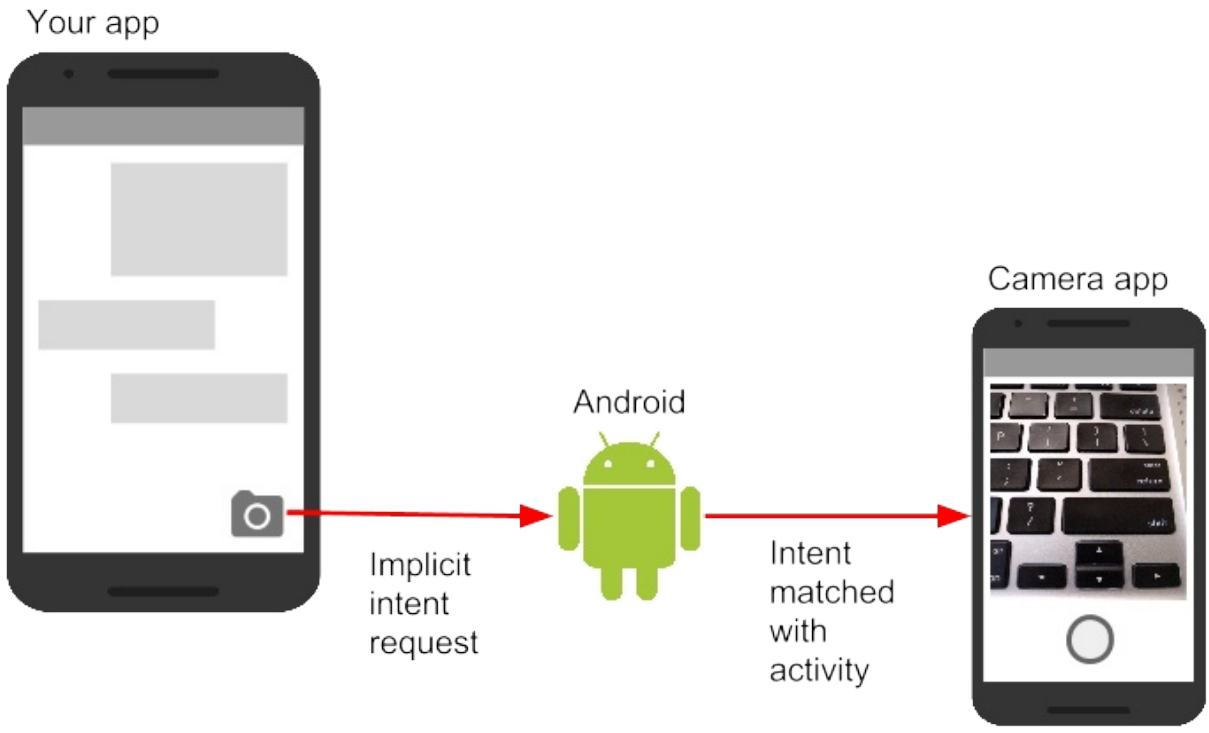
In a previous chapter you learned about intents, and how to launch specific activities in your app with *explicit intents*. In this chapter you'll learn how to send and receive *implicit* intents, where you declare a general action to perform in the intent, and the system matches your request with a specific activity. Additionally, you'll learn more about Android *tasks*, and how you can configure your apps to associate new activities with different tasks.

About implicit intents

In an earlier chapter you learned about explicit intents, where you can start one activity from another by specifying the class name of that activity. This is the most basic way to use intents, to start an activity or other app component and pass data to it (and sometimes pass data back.)

A more flexible use of intents is the *implicit intent*. With implicit intents you do not specify the exact activity (or other component) to run—instead, you include just enough information in the intent about the task you want to perform. The Android system matches the information in your request intent with activities available on the device that can perform that

task. If there's only one activity that matches, that activity is launched. If there are multiple matching activities, the user is presented with an app chooser that enables them to pick which app they would like to perform the task.



For example, you have an app that lists available snippets of video. If the user touches an item in the list, you want to play that video snippet. Rather than implementing an entire video player in your own app, you can launch an intent that specifies the task as "play an object of type video." The Android system then matches your request with an activity that has registered itself to play objects of type video.

Activities register themselves with the system as being able to handle implicit intents with intent *filters*, declared in the Android manifest. For example, the main activity (and only the main activity) for your app has an intent filter that declares it the main activity for the launcher category. This intent filter is how the Android system knows to start that specific activity in your app when the user taps the icon for your app on the device home screen.

Intent actions, categories, and data

Implicit intents, like explicit intents, are instances of the Intent class. In addition to the parts of an intent you learned about in an earlier chapter (such as the intent data and intent extras), these fields are used by implicit intents:

- The intent *action*, which is the generic action the receiving activity should perform. The available intent actions are defined as constants in the Intent class and begin with the word ACTION_. A common intent action is ACTION_VIEW, which you use when you have some information that an activity can show to the user, such as a photo to view in a gallery app, or an address to view in a map app. You can specify the action for an intent in the intent constructor, or with the setAction() method.
- An intent *category*, which provides additional information about the category of component that should handle the intent. Intent categories are optional, and you can add more than one category to an intent. Intent categories are also defined as constants in the Intent class and begin with the word CATEGORY_. You can add categories to the intent with the addCategory() method.
- The data *type*, which indicates the MIME type of data the activity should operate on. Usually, this is inferred from the URI in the intent data field, but you can also explicitly define the data type with the setType() method.

Intent actions, categories, and data types are used both by the Intent object you create in your sending activity, as well as in the intent filters you define in the Android manifest for the receiving activity. The Android system uses this information to match an implicit intent request with an activity or other component that can handle that intent.

Sending implicit intents

Starting activities with implicit intents, and passing data between those activities, works much the same way as it does for explicit intents:

1. In the sending activity, create a new Intent object.
2. Add information about the request to the Intent object, such as data or extras.
3. Send the intent with startActivity() (to just start the activity) or startActivityforResult() (to start the activity and expect a result back).

When you create an implicit Intent object, you:

- Do not specify the specific activity or other component to launch.
- Add an intent action or intent categories (or both).
- Resolve the intent with the system before calling startActivity() or startActivityforResult().
- Show an app chooser for the request (optional).

Create implicit Intent objects

To use an implicit intent, create an Intent object as you did for an explicit intent, only without the specific component name.

```
Intent sendIntent = new Intent();
```

You can also create the Intent object with a specific action:

```
Intent sendIntent = new Intent(Intent.ACTION_VIEW);
```

Once you have an Intent object you can add other information (category, data, extras) with the various Intent methods. For example, this code creates an implicit Intent object, sets the intent action to ACTION_SEND, defines an intent extra to hold the text, and sets the type of the data to the MIME type "text/plain".

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");
```

Resolve the activity before starting it

When you define an implicit intent with a specific action and/or category, there is a possibility that there won't be *any* activities on the device that can handle your request. If you just send the intent and there is no appropriate match, your app will crash.

To verify that an activity or other component is available to receive your intent, use the resolveActivity() method with the system package manager like this:

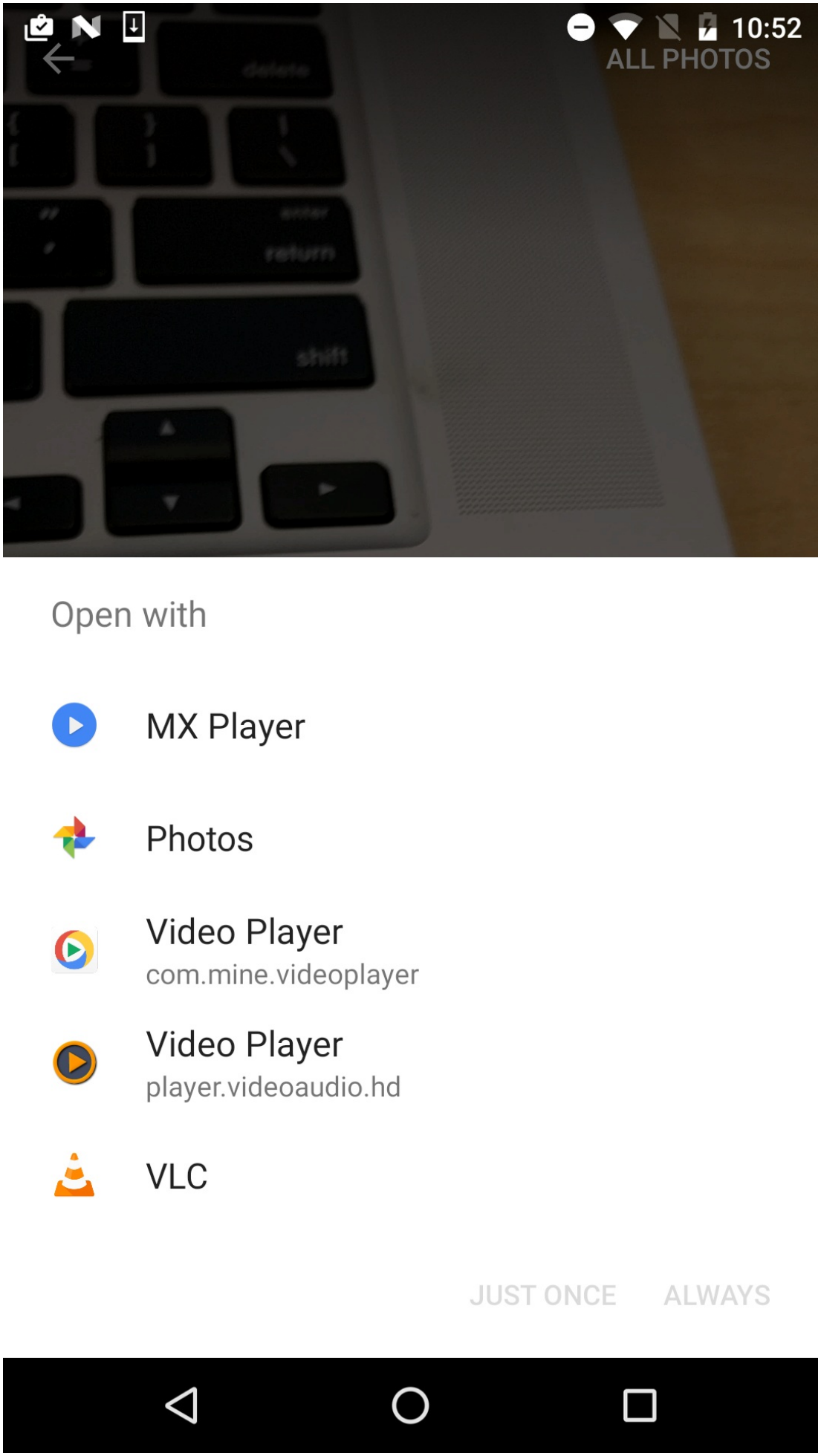
```
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

If the result of resolveActivity() is not null, then there is at least one app available that can handle the intent, and it's safe to call startActivity(). Do not send the intent if the result is null.

If you have a feature that depends on an external activity that may or may not be available on the device, a best practice is to test for the availability of that external activity before the user tries to use it. If there is no activity that can handle your request (that is, resolveActivity() returns null), disable the feature or provide the user an error message for that feature.

Show the app chooser

To find an activity or other component that can handle your intent requests, the Android system matches your implicit intent with an activity whose intent filters indicate that they can perform that action. If there are multiple apps installed that match, the user is presented with an app chooser that lets them select which app they want to use to handle that intent.


10:52
ALL PHOTOS
Open with
MX Player
Photos
Video Player
com.mine.videoplayer
Video Player
player.videoaudio.hd
VLC
JUST ONCE
ALWAYS

In many cases the user has a preferred app for a given task, and they will select the option to always use that app for that task. However, if multiple apps can respond to the intent and the user might want to use a different app each time, you can choose to explicitly show a chooser dialog every time. For example, when your app performs a "share this" action with the ACTION_SEND action, users may want to share using a different app depending on the current situation.

To show the chooser, you create a wrapper intent for your implicit intent with the createChooser() method, and then resolve and call startActivity() with that wrapper intent. The createChooser() method also requires a string argument for the title that appears on the chooser. You can specify the title with a string resource as you would any other string.

For example:

```
// The implicit Intent object
Intent sendIntent = new Intent(Intent.ACTION_SEND);
// Always use string resources for UI text.
String title = getResources().getString(R.string.chooser_title);
// Create the wrapper intent to show the chooser dialog.
Intent chooser = Intent.createChooser(sendIntent, title);
// Resolve the intent before starting the activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

Receiving implicit intents

If you want an activity in your app to respond to implicit intents (from your own app or other apps), declare one or more intent filters in the Android manifest. Each intent filter specifies the type of intents it accepts based on the intent's action, data, and category. The system will deliver an implicit intent to your app component only if that intent can pass through one of your intent filters.

Note: An explicit intent is always delivered to its target, regardless of any intent filters the component declares. Conversely, if your activities do not declare any intent filters, they can only be launched with an explicit intent.
Once your activity is successfully launched with an implicit intent you can handle that intent and its data the same way you did an explicit intent, by:

1. Getting the Intent object with getIntent().
2. Getting intent data or extras out of that intent.
3. Performing the task the intent requested.
4. Returning data to the calling activity with another intent, if needed.

Intent filters

Define intent filters with one or more `<intent-filter>` elements in the app's manifest file, nested in the corresponding `<activity>` element. Inside `<intent-filter>`, specify the type of intents your activity can handle. The Android system matches an implicit intent with an activity or other app component only if the fields in the Intent object match the intent filters for that component.

An intent filter may contain these elements, which correspond to the fields in the Intent object described above:

- `<action>`: The intent action.
- `<data>`: The type of data accepted, including the MIME type or other attributes of the data URI (such as scheme, host, port, path, and so no).
- `<category>`: The intent category.

For example, the main activity for your app includes this `<intent-filter>` element, which you saw in an earlier chapter:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

This intent filter has the action MAIN and the category LAUNCHER. The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity). Only the main activity for your app should have this intent filter.

Here's another example for an implicit intent to share a bit of text. This intent filter matches the implicit intent example from the previous section:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

You can specify more than one action, data, or category for the same intent filter, or have multiple intent filters per activity to handle different kinds of intents.

The Android system tests an implicit intent against an intent filter by comparing the parts of that intent to each of the three intent filter elements (action, category, and data). The intent must pass all three tests or the Android system won't deliver the intent to the component. However, because a component may have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another filter.

Actions

An intent filter can declare zero or more `<action>` elements for the intent action. The action is defined in the name attribute, and consists of the string "android.intent.action." plus the name of the intent action, minus the ACTION_ prefix. So, for example, an implicit intent with the action ACTION_VIEW matches an intent filter whose action is `android.intent.action.VIEW.`

For example, this intent filter matches either ACTION_EDIT and ACTION_VIEW:

```
<intent-filter>
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.VIEW" />
    ...
</intent-filter>
```

To get through this filter, the action specified in the incoming Intent object must match at least one of the actions. You must include at least one intent action for an incoming implicit intent to match.

Categories

An intent filter can declare zero or more `<category>` elements for intent categories. The category is defined in the name attribute, and consists of the string "android.intent.category." plus the name of the intent category, minus the CATEGORY prefix.

For example, this intent filter matches either CATEGORY_DEFAULT and CATEGORY_BROWSABLE:

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    ...
</intent-filter>
```

Note that all activities that you want to accept implicit intents must include the android.intent.category.DEFAULT intent-filter. This category is applied to all implicit Intent objects by the Android system.

Data

An intent filter can declare zero or more `<data>` elements for the URI contained in the intent data. As the intent data consists of a URI and (optionally) a MIME type, you can create an intent filter for various aspects of that data, including:

- URI Scheme
- URI Host
- URI Path
- Mime type

For example, this intent filter matches data intents with a URI scheme of http and a MIME type of either "video/mpeg" or "audio/mpeg".

```
<intent-filter>
    <data android:mimeType="video/mpeg" android:scheme="http" />
    <data android:mimeType="audio/mpeg" android:scheme="http" />
    ...
</intent-filter>
```

Sharing data with ShareCompat.IntentBuilder

Share actions are an easy way for users to share items in your app with social networks and other apps. Although you can build a share action in your own app using implicit intents with the ACTION_SEND action, Android provides the ShareCompat.IntentBuilder helper class to easily implement sharing in your app.

Note: For apps that target Android releases after API 14, you can use the ShareActionProvider class for share actions instead of ShareCompat.IntentBuilder. The ShareCompat class is part of the V4 support library, and allows you to provide share actions in apps in a backward-compatible fashion. ShareCompat provides a single API for sharing on both old and new Android devices. You'll learn more about the Android support libraries in a later chapter.
With the ShareCompat.IntentBuilder class you do not need to create or send an implicit intent for the share action. Use the methods in ShareCompat.IntentBuilder to indicate the data you want to share as well as any additional information. Start with the from() method to create a new intent builder, add other methods to add more data, and end with the startChooser() method to create and send the intent. You can chain the methods together like this:

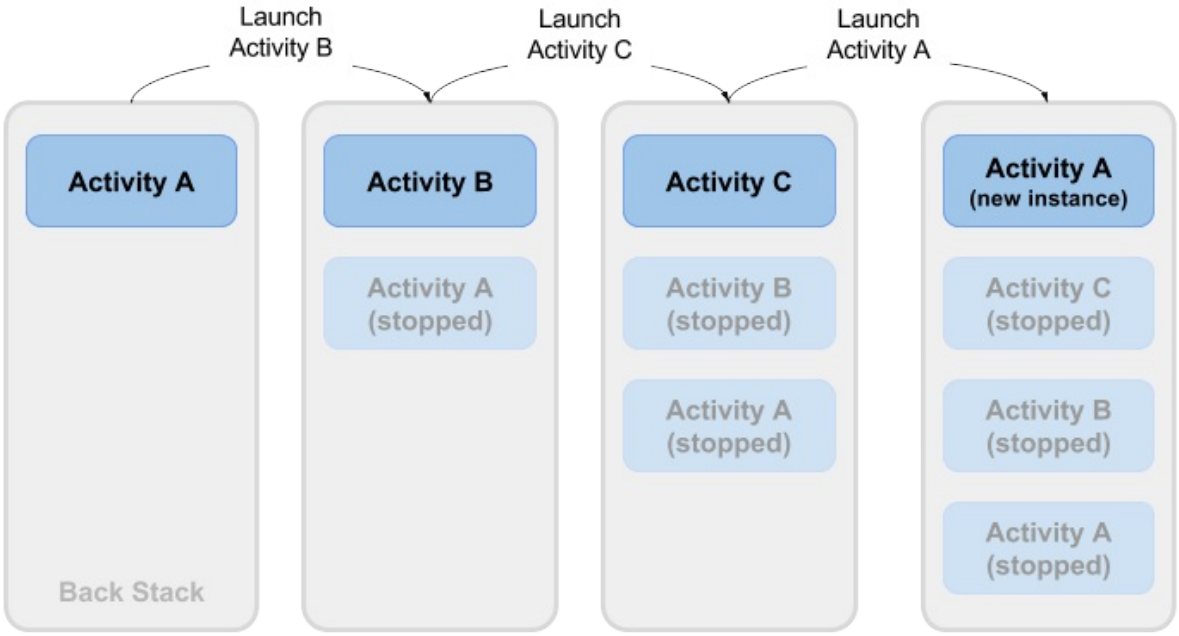
```
ShareCompat.IntentBuilder
    .from(this)         // information about the calling activity
    .setType(mimeType)  // mime type for the data
    .setChooserTitle("Share this text with: ") //title for the app chooser
    .setText(txt)       // intent data
    .startChooser();    // send the intent
```

Managing tasks and activities

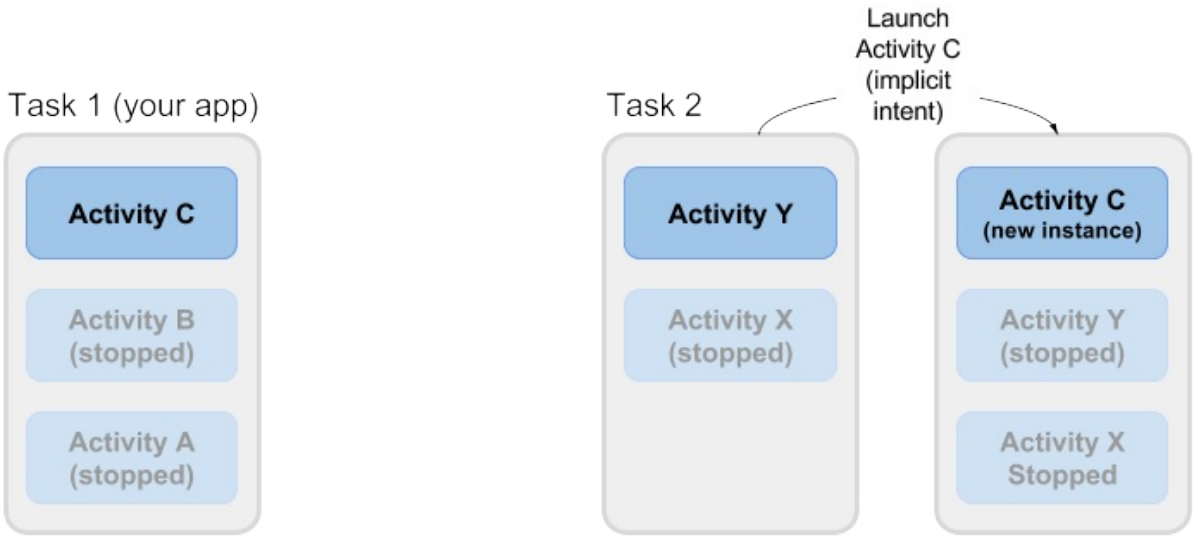
In a previous chapter you learned about tasks and the back stack, in which the task for your app contains its own stack for the activities the user has visited while they use your app. As the user navigates around your app, activity instances for that task are pushed and popped from the stack for that task.

Most of the time the user's navigation from activity to activity and back again through the stack is straightforward. Depending on the design and navigation of your app there may be complications, especially with activities that are started from other apps and other tasks.

For example, say you have an app with three activities: A, B, and C. A launches B with an intent, and B launches C. C, in turn sends an intent to launch A. In this case the system creates a *second instance* of A on the top of the stack, rather than bringing the already-running instance to the foreground. Depending on how you implement your activities, the two instances of A can get out of sync and provide a confusing experience for the user as they navigate back through the stack.



Or, say your activity C can be launched from a second app with an implicit intent. The user runs the second app, which has its own task and its own back stack. If that app uses an implicit intent to launch your activity C, a new instance of C is created and placed on the back stack for that second app's task. Your app still has its own task, its own back stack, and its own instance of C.



Much of the time the Android's default behavior for tasks and activities works fine and you don't have to worry about how your activities are associated with tasks, or how they exist in the back stack. If you want to change the normal behavior, Android provides a number of ways to manage tasks and the activities within those tasks, including:

- Activity launch modes, to determine how an activity should be launched.
- Task affinities, which indicate which task a launched activity belongs to.

Activity Launch Modes

Use activity launch modes to indicate how new activities should be treated when they're launched—that is, if they should be added to the current task, or launched into a new task. Define launch modes for the activity with attributes on the `<activity>` element of the Android manifest, or with flags set on the intent that starts that activity.

Activity attributes

To define a launch mode for an activity add the android:launchMode attribute to the `<activity>` element in the Android manifest. This example uses a launch mode of "standard", which is the default.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:launchMode="standard">
   ...
</activity>
```

There are four launch modes available as part of the `<activity>` element:

- "standard" (the default): New activities are launched and added to the back stack for the current task. An activity can be instantiated multiple times, a single task can have multiple instances of the same activity, and multiple instances can belong to different tasks.
- "singleTop": If an instance of an activity exists at the top of the back stack for the current task and an intent request for that activity arrives, Android routes that intent to the existing activity instance rather than creating a new instance. A new activity is still instantiated if there is an existing activity anywhere in the back stack other than the top.
- "singleTask": When the activity is launched the system creates a new task for that activity. If another task already exists with an instance of that activity, the system routes the intent to that activity instead.
- "singleInstance": Same as single task, except that the system doesn't launch any other activities into the task holding the activity instance. The activity is always the single and only member of its task.

The vast majority of apps will only use the standard or single top launch modes. See the launchMode attribute documentation for more detailed information on launch modes.

Intent flags

Intent flags are options that specify how the activity (or other app component) that receives the intent should handle that intent. Intent flags are defined as constants in the Intent class and begin with the word FLAG_. You add intent flags to an Intent object with setFlag() or addFlag().

Three specific intent flags are used to control activity launch modes, either in conjunction with the launchMode attribute or in place of it. Intent flags always take precedence over the launch mode in case of conflicts.

- FLAG_ACTIVITY_NEW_TASK: start the activity in a new task. This is the same behavior as the singleTask launch mode.
- FLAG_ACTIVITY_SINGLE_TOP: if the activity to be launched is at the top of the back stack, route the intent to that existing activity instance. Otherwise create a new activity instance. This is the same behavior as the singleTop launch mode.
- FLAG_ACTIVITY_CLEAR_TOP: If an instance of the activity to be launched already exists in the back stack, destroy any other activities on top of it and route the intent to that existing instance. When used in conjunction with FLAG_ACTIVITY_NEW_TASK, this flag locates any existing instances of the activity in any task and brings it to the foreground.

See the Intent class for more information about other available intent flags.

Handle new intents

When the Android system routes an intent to an existing activity instance, the system calls the onNewIntent() callback method (usually just before the onResume() method). The onNewIntent() method includes an argument for the new intent that was routed to the activity. Override the onNewIntent() method in your class to to handle the information from that new intent.

Note that the getIntent() method—to get access to the intent that launched the activity—**always** retains the original intent that launched the activity instance. Call setIntent() in the onNewIntent() method:

```
@Override
public void onNewIntent(Intent intent) {
    super.onNewIntent(intent);

    // Use the new intent, not the original one
    setIntent(intent);
}
```

Any call to getIntent() after this returns the new intent.

Task Affinities

Task affinities indicate which task an activity prefers to belong to when that activity instance is launched. By default all activities belong to the app that launched them. Activities from outside an app launched with implicit intents belong to the app that sent the implicit intent.

To define a task affinity, add the android:taskAffinity attribute to the `<activity>` element in the Android manifest. The default task affinity is the package name for the app (declared in . The new task name should be unique and different from the package name. This example uses "com.example.android.myapp.newtask" for the affinity name.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:taskAffinity="com.example.android.myapp.newtask">
   ...
</activity>
```

Task affinities are often used in conjunction with the singleTask launch mode or the FLAG_ACTIVITY_NEW_TASK intent flag to place a new activity in its own named task. If the new task already exists, the intent is routed to that task and that affinity.

Another use of task affinities is reparenting, which enables a task to move from the activity in which it was launched to the activity it has an affinity for. To enable task reparenting, add a task affinity attribute to the `<activity>` element and set android:allowTaskReparenting to true.

```
<activity
   android:name=".SecondActivity"
   android:label="@string/activity2_name"
   android:parentActivityName=".MainActivity"
   android:taskAffinity="com.example.android.myapp.newtask"
   android:allowTaskReparenting="true" >
   ...
</activity>
```

Related Practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Activities and Implicit Events

Learn More

- Activities (API Guide)
- Activity (API Reference)
- Intents and Intent Filters
- Intent (API Reference)
- `<intent-filter>`
- Allowing Other Apps to Start Your Activity
- ShareCompat.IntentBuilder (API Reference)
- Uri (API Reference)
- Google Maps Intents
- Tasks and Back Stack
- `<activity>`
- Manipulating Android tasks and back stack
- Android task affinity explanation
- Understand Android Activity's launchMode

3.1: The Android Studio Debugger

Contents:

- Introduction
- About debugging
- Running the debugger
- Using Breakpoints
- Stepping through code
- Viewing execution stack frames
- Inspecting and modifying variables
- Setting watches
- Evaluating expressions
- More tools for debugging
- Trace logging and the Android manifest
- Related practical
- Learn more

In this chapter you'll learn about debugging your apps in Android Studio.

About debugging

Debugging is the process of finding and fixing errors (bugs) or unexpected behavior in your code. All code has bugs, from incorrect behavior in your app, to behavior that excessively consumes memory or network resources, to actual app freezing or crashing.

Bugs can result for many reasons:

- Errors in your design or implementation.
- Android framework limitations (or bugs).
- Missing requirements or assumptions for how the app should work.
- Device limitations (or bugs)

Use the debugging, testing, and profiling capabilities in Android Studio to help you reproduce, find, and resolve all of these problems. Those capabilities include:

- The Android monitor (logcat)
- The Android Studio debugger
- Testing frameworks such as JUnit or Espresso
- Dalvik Debug Monitor Server (DDMS), to track resource usage

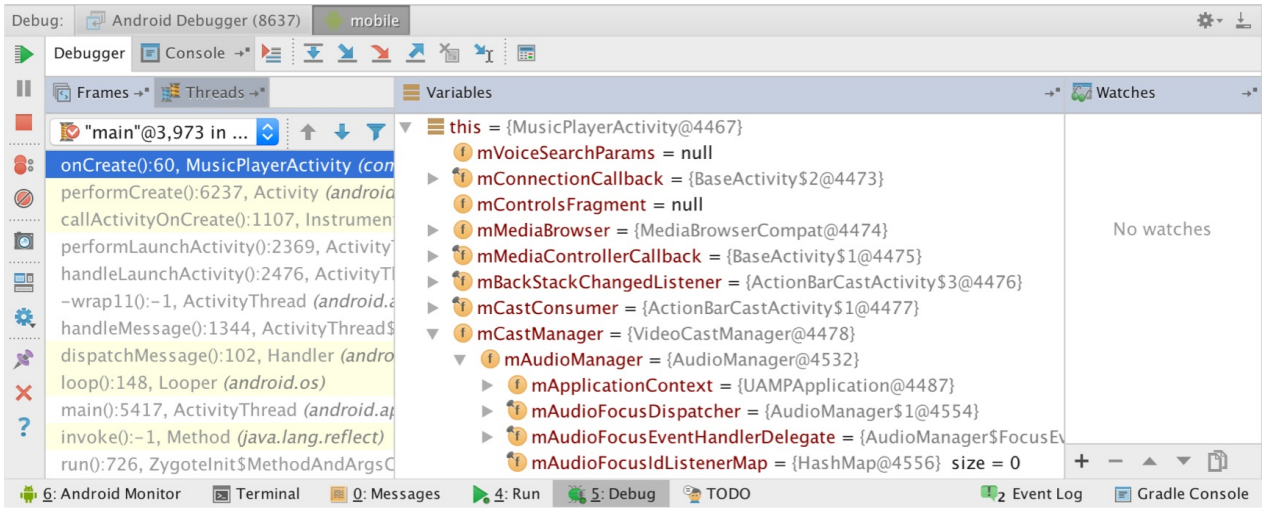
In this chapter you'll learn how to debug your app with the Android Studio debugger, set and view breakpoints, step through your code, and examine variables.

Running the debugger

Running an app in debug mode is similar to just running that app. You can either run an app in debug mode, or attach the debugger to an already-running app.

Run your app in debug mode

To start debugging, click Debug in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, then runs it and opens the Debug window.



Debug a running app

If your app is already running on a device or emulator, start debugging that app with these steps:

1. Select **Run > Attach debugger to Android process** or click the **Attach** icon in the toolbar.
2. In the **Choose Process** dialog, select the process to which you want to attach the debugger.

   By default, the debugger shows the device and app process for the current project, as well as any connected hardware devices or virtual devices on your computer. Select Show all processes to show all processes on all devices.

3. Click **OK**. The Debug window appears as before.

Resume or Stop Debugging

To resume executing an app after debugging it, select **Run > Resume Program** or click the Resume icon.

To stop debugging your app, select **Run > Stop** or click the Stop icon in the toolbar.

Using Breakpoints

A breakpoint is a place in your code where you want to pause the normal execution of your app to perform other actions such as examining variables or evaluating expressions, or executing your code line by line to determine the causes of runtime errors.

Add breakpoints

To add a breakpoint to a line in your code, use these steps:

1. Locate the line of code where you want to pause execution.
2. Click in the left gutter of the editor window at that line, next to the line numbers. A red dot appears at that line, indicating a breakpoint.

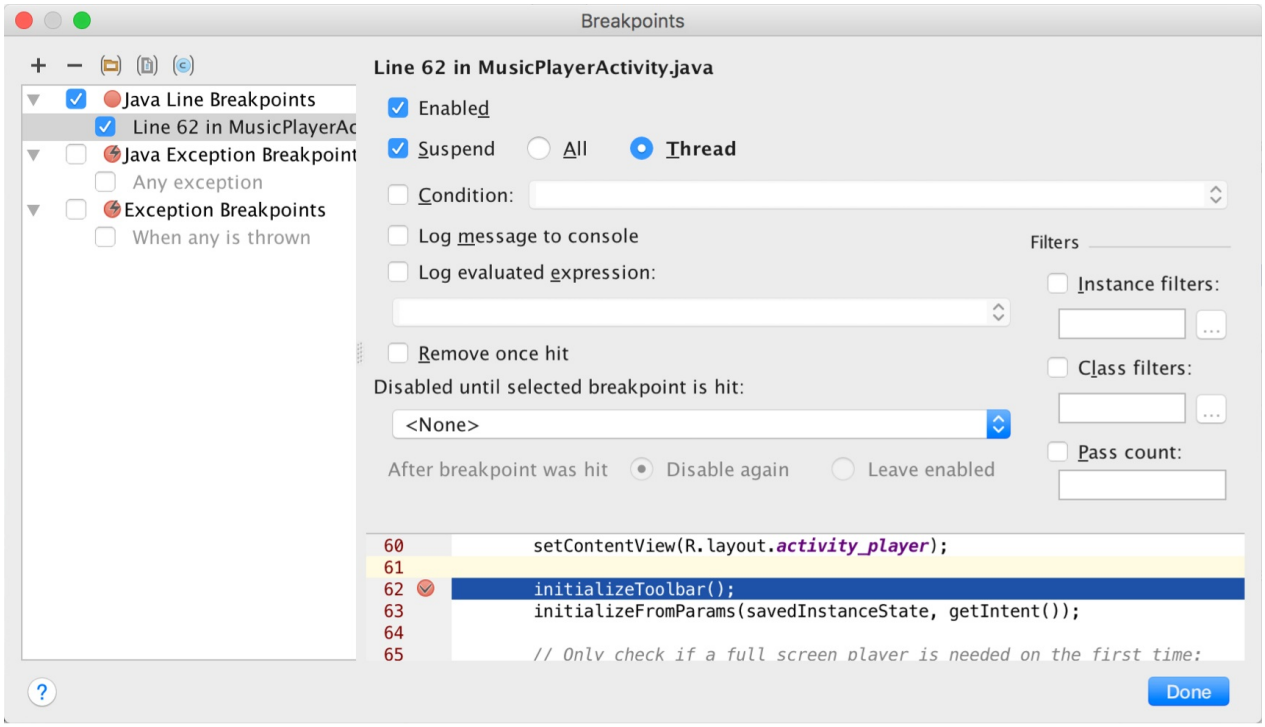
   You can also use **Run > Toggle Line Breakpoint** or Control-F8 (Command-F8 on OS X) to set a breakpoint at a line.

If your app is already running, you don't need to update it to add the breakpoint.

When your code execution reaches the breakpoint, Android Studio pauses execution of your app. You can then use the tools in the Android debugger to view the state of the app and debug that app as it runs.

View and configure breakpoints

To view all the breakpoints you've set and configure breakpoint settings, click the View Breakpoints icon on the left edge of the debugger window. The Breakpoints window appears.



In this window all the breakpoints you have set appear in the left pane, and you can enable or disable each breakpoint with the check boxes. If a breakpoint is disabled, Android Studio does not pause your app when execution reaches that breakpoint.

Select a breakpoint from the list to configure its settings. You can configure a breakpoint to be disabled at first and have the system enable it after a different breakpoint is encountered. You can also configure whether a breakpoint should be disabled after it has been reached.

To set a breakpoint for any exception, select Exception Breakpoints in the list of breakpoints.

Disable (Mute) all breakpoints

Disabling a breakpoint enables you to temporarily "mute" that breakpoint without removing it from your code. If you remove a breakpoint altogether you also lose any conditions or other features you created for that breakpoint, so disabling it can be a better choice.

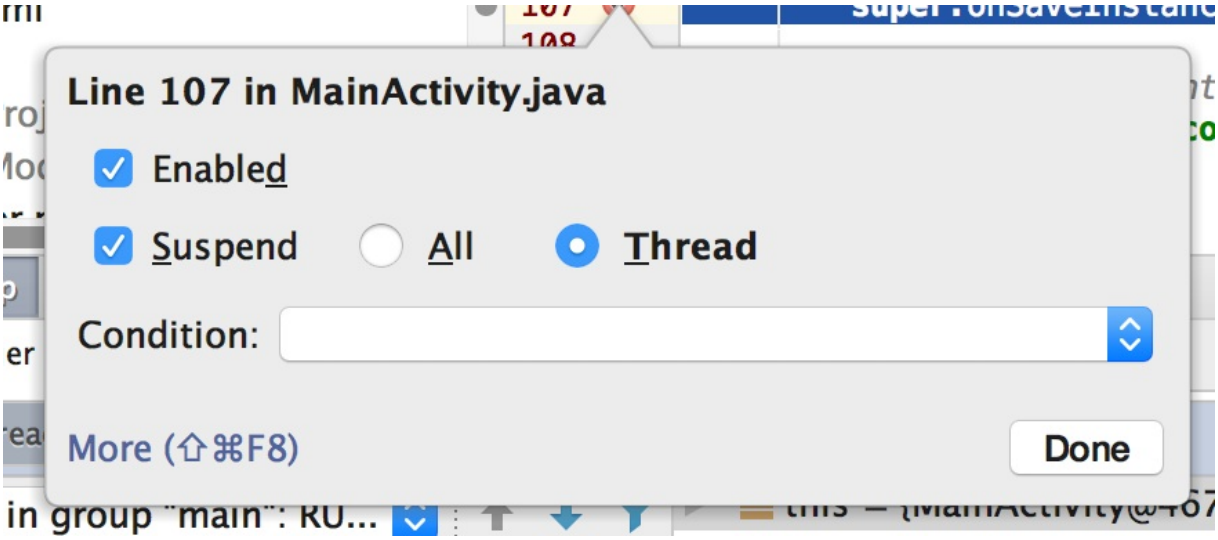
To mute all breakpoints, click the **Mute Breakpoints** icon. Click the icon again to enable (unmute) all breakpoints.

Use conditional breakpoints

Conditional breakpoints are breakpoints that only stop execution of your app if the test in the condition is true. To define a test for a conditional breakpoint, use these steps:

1. Right click on a breakpoint icon, and enter a test in the Condition field.

You can also use the Breakpoints window to enter a breakpoint condition.



The test you enter in this field can be any Java expression as long as it returns a boolean value. You can use variable names from your app as part of the expression.

2. Run your app in debug mode. Execution of your app stops at the conditional breakpoint, if the condition evaluates to true.

Stepping through code

After your app's execution has stopped because a breakpoint has been reached, you can execute your code from that point one line at a time with the Step Over, Step Into, and Step Out functions.

To use any of the step functions:

1. Begin debugging your app. Pause the execution of your app with a breakpoint.

   Your app's execution stops, and the debugger shows the current state of the app. The current line is highlighted in your code.

2. Click the **Step Over** icon, select **Run > Step Over,** or type F8.

   Step Over executes the next line of the code in the current class and method, executing all of the method calls on that line and remaining in the same file.

3. Click the **Step Into** icon, select **Run > Step Into,** or type F7.

   Step Into jumps into the execution of a method call on the current line (versus just executing that method and remaining on the same line). The **Frames** view (which you'll learn about in the next section) updates to show the new stack frame (the new method). If the method call is contained in another class, the file for that class is opened and the current line in that file is highlighted. You can continue stepping over lines in this new method call, or step deeper into other methods.

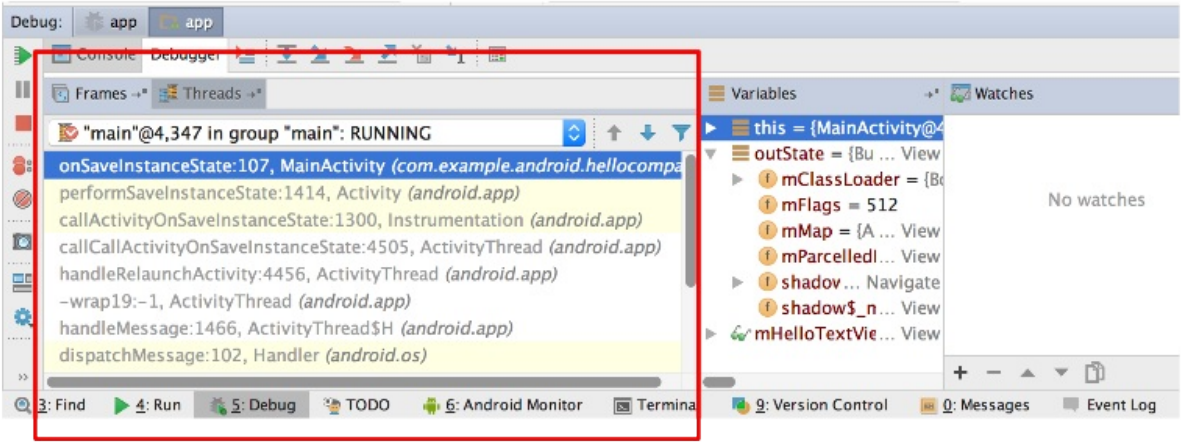
4. Click the **Step Out** icon, select **Run > Step Out,** or type Shift-F8.

   Step Out finishes executing the current method and returns to the point where that method was called.

5. To resume normal execution of the app, select **Run > Resume Program** or click the Resume icon.

Viewing execution stack frames

The **Frames** view of the debugger window allows you to inspect the execution stack and the specific frame that caused the current breakpoint to be reached.



The execution stack shows all the classes and methods (frames) that are being executed up to this point in the app, in reverse order (most recent frame first). As execution of a particular frame finishes, that frame is popped from the stack and execution returns to the next frame.

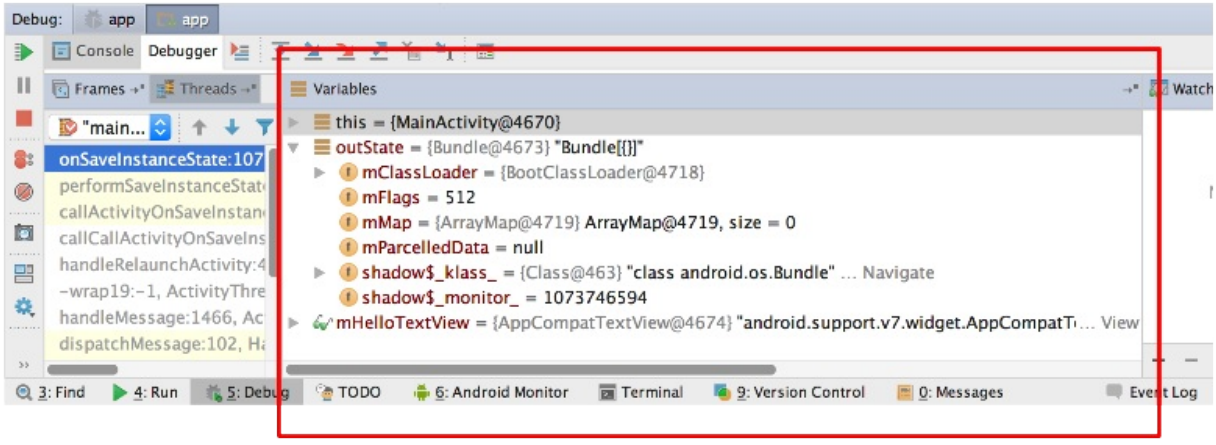
Clicking a line for a frame in the Frames view opens the associated source in the editor and highlights the line where that frame was initially executed. The Variables and Watches views also update to reflect the state of the execution environment when that frame was last entered.

Inspecting and modifying variables

The **Variables** view of the debugger window allows you to inspect the variables available at the current stack frame when the system stops your app on a breakpoint. Variables that hold objects or collections such as arrays can be expanded to view their components.

The **Variables** pane also allows you to evaluate expressions on the fly using static methods and/or variables available within the selected frame.

If the Variables view is not visible, click the **Restore Variables View** icon .



To modify variables in your app as it runs:

1. Right-click any variable in the Variables view, and select **Set Value**. You can also use F2.
2. Enter a new value for the variable, and type Return.

The value you enter must be of the appropriate type for that variable, or Android Studio returns a "type mismatch" error.

Setting watches

The **Watches** view provides similar functionality to the Variables view except that expressions added to the Watches pane persist between debugging sessions. Add watches for variables and fields that you access frequently or that provide state that is helpful for the current debugging session.

To use watches:

1. Begin debugging your app.
2. In the Watches pane, click the plus (+) button.

   In the text box that appears, type the name of the variable or expression you want to watch and then press Enter.

Remove an item from the Watches list by selecting the item and then clicking the minus (-) button.

Change the order of the elements in the Watches list by selecting an item and then clicking the up or down icons.

Evaluating expressions

Use Evaluate Expression to explore the state of variables and objects in your app, including calling methods on those objects. To evaluate an expression:

1. Click the Evaluate Expression icon, or select **Run > Evaluate Expression**. You can also right-click on any variable and choose Evaluate Expression.

   The Evaluate Expression window appears.

2. Enter any expression into the Expression window and click **Evaluate**.

   The Evaluate Expression window updates with the result of the execution. Note that the result you get from evaluating an expression is based on the app's current state. Depending on the values of the variables in your app at the time you evaluate expressions, you may get different results. Changing the values of variables in your expressions also changes the current running state of the app.

More tools for debugging

Android Studio and the Android SDK include a number of other tools to help you find and correct issues in your code. These tools include:

- System log (logcat). As you've learned in previous lessons, you can use the Log class to send messages to the Android system log, and view those messages in Android Studio.
  - To write log messages in your code, use the Log class. Log messages help you understand the execution flow by collecting the system debug output while you interact with your app. Log messages can tell you what part of your application failed. For more information about logging, see Reading and Writing Logs.
- Tracing and Logging. Analyzing traces allows you to see how much time is spent in certain methods, and which ones are taking the longest times.
  - To create the trace files, include the Debug class and and call one of the startMethodTracing() methods. In the call, you specify a base name for the trace files that the system generates. To stop tracing, call stopMethodTracing(). These methods start and stop method tracing across the entire virtual machine. For example, you could call startMethodTracing() in your activity's onCreate() method, and call stopMethodTracing() in that activity's onDestroy() method.
- The Android Debug Bridge (ADB). ADB is a command-line tool that lets you communicate with an emulator instance or

connected Android-powered device.
- Dalvik Debug Monitor Server (DDMS). The DDSM tool provides port-forwarding services, screen capture, thread and heap information, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, and more.
- CPU and memory monitors. Android Studio includes a number of monitors to help you visualize the behavior and performance of your app.
- Screenshot and video capture.

Trace logging and the Android manifest

There are multiple types of debugging available to you beyond setting breakpoints and stepping through code. You can also use logging and tracing to find issues with your code. When you have a trace log file (generated by adding tracing code to your application or by DDMS), you can load the log files in Traceview, which displays the log data in two panels:

- A timeline panel -- describes when each thread and method started and stopped
- A profile panel -- provides a summary of what happened inside a method

Likewise, you can set android:debuggable in the `<application>` tag of the Android Manifest to `"true"`, which sets whether or not the application can be debugged, even when running on a device in user mode. By default, this value is set to `"false"`.

You can create and configure build types in the module-level build.gradle file inside the android {} block. When you create a new module, Android Studio automatically creates the debug and release build types for you. Although the debug build type doesn't appear in the build configuration file, Android Studio configures it with debuggable true. This allows you to debug the app on secure Android devices and configures APK signing with a generic debug keystore. You can add the the debug build type to your configuration if you want to add or change certain settings.

All these changes made for debugging must be removed from your code before release because they can impact the execution and performance production code.

When you prepare your app for release, you must remove all the extra code in your source files that you wrote for testing purposes.

In addition to prepping the code itself, there are a few other tasks you need to complete in order to get your app ready to publish. These include:

- Removing logging statements
- Remove any calls to show Toasts
- Disable debugging in the Android manifest by either:
  - Removing android:debuggable attribute from `<application>` tag
  - Or setting android:debuggable attribute to false

Remove all debug tracing calls from your source code files such as startMethodTracing() and stopMethodTracing().

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Using the Debugger

Learn more

- Debug Your App
- Write and View Logs
- Analyze a Stack Trace

- Android Monitor
- Using DDMS
- Android Debug Bridge
- Android Monitor Overview
- Create and Edit Run/Debug Configurations
- Debugging and Testing in Android Studio (video)

3.2: Testing your App

Contents:

- Introduction
- About testing
- Setting up testing
- Creating and running unit tests
- Related Practical
- Learn More

In this chapter you'll get an overview of Android testing, and about creating and running local unit tests in Android Studio with JUnit.

About testing

Even though you have an app that compiles and runs and looks the way you want it to on different devices, you must make sure that your app will behave the way you expect it to in every situation, especially as your app grows and changes. Even if you try to manually test your app every time you make a change — a tedious prospect at best — you might miss something or not anticipate what end users might do with your app to cause it to fail.

Writing and running tests is a critical part of the software development process. "Test-Driven Development" (TDD) is a popular software development philosophy that places tests at the core of all software development for an application or service.This does not negate the need for further testing, it merely gives you a solid baseline to work with.

Testing your code can help you catch issues early in development—when they are the least expensive to address — and improve the robustness of your code as your app gets larger and more complex. With tests in your code, you can exercise small portions of your app in isolation, and in an automatable and repeatable manner. Because….the code you write to test your app doesn't end up in the production version of your app; it lives only on your development machine, alongside your app's code in Android Studio.

Types of tests

Android supports several different kinds of tests and testing frameworks. Two basic forms of testing Android Studio supports are local unit tests and instrumented tests.

Local unit tests are tests that are compiled and run entirely on your local machine with the Java Virtual Machine (JVM). Use local unit tests to test the parts of your app (such as the internal logic) that do not need access to the Android framework or an Android device or emulator, or those for which you can create fake ("mock" or stub) objects that pretend to behave like the framework equivalents.

Instrumented tests are tests that run on an Android device or emulator. These tests have access to the Android framework and to Instrumentation information such as the app's Context. You can use instrumented tests for unit testing, user interface (UI) testing, or integration testing, making sure the components of your app interact correctly with other apps. Most commonly, you use instrumented tests for UI testing, which allows you to test that your app behaves correctly when a user interacts with your app's activities or enters a specific input.

For most forms of user interface testing, you use the Espresso framework, which allows you to write automated UI tests. You'll learn about instrumented tests and Espresso in a later chapter.

Unit Testing

Unit tests should be the fundamental tests in your app testing strategy. By creating and running unit tests against your code, you can verify that the logic of individual functional code areas or units is correct. Running unit tests after every build helps you catch and fix problems introduced by code changes to your app.

A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way. Create unit tests when you need to verify the logic of specific code in your app. For example, if you are unit testing a class, your test might check that the class is in the right state. For a method, you might test its behavior for different values of its parameters, especially null.Typically, the unit of code is tested in isolation; your test monitors changes to that unit only. A mocking framework such as Mockito can be used to isolate your unit from its dependencies.You can also write your unit tests for Android in JUnit 4, a common unit testing framework for Java code.

The Android Testing Support Library

The Android Testing Support Library provides the infrastructure and APIs for testing Android apps, including support for JUnit 4. With the testing support library you can build and run test code for your apps.

You may already have the Android Testing Support Library installed with Android Studio. To check for the Android Support Repository, follow these steps:

1. In Android Studio choose **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab, and look for the Support Repository.
3. If necessary, update or install the library.

The Android Testing Support Library classes are located under the android.support.test package. There are also older testing APIs in android.test. You should use the support libraries first, when given a choice between the support libraries and the older APIs, as the support libraries help build and distribute tests in a cleaner and more reliable fashion than directly coding against the API itself.

Setting up testing

To prepare your project for testing in Android Studio, you need to:

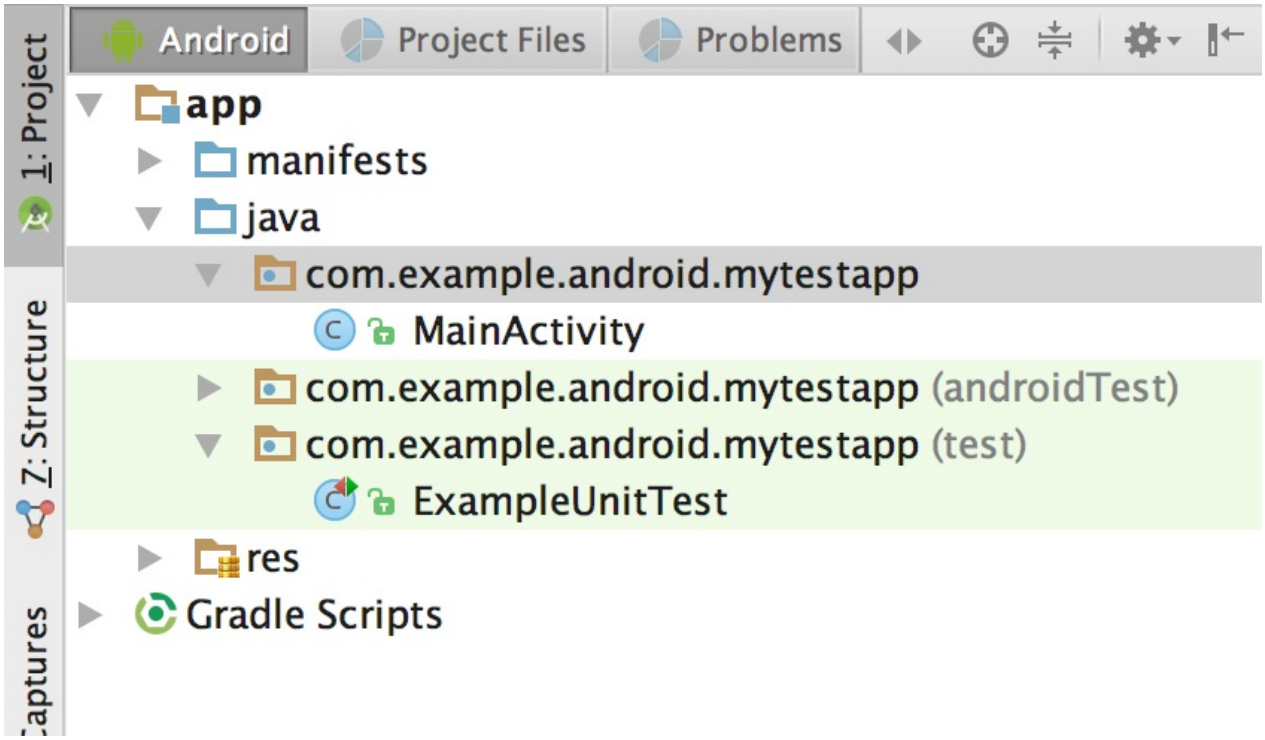
- Organize your tests in a *source set*.
- Configure your project's gradle dependencies to include testing-related APIs.

Android Studio source sets

Source sets are a collection of related code in your project that are for different build targets or other "flavors" of your app. When Android Studio creates your project, it creates three source sets for you:

- The main source set, for your app's code and resources.
- The **test** source set, for your app's local unit tests.
- The **androidTest** source set, for Android instrumented tests.

Source sets appear in the Android Studio Android view under the package name for your app. The main source set includes just the package name. The test and androidTest source sets have the package name followed by (test) or (androidTest), respectively.



These source sets correspond to folders in the src directory for your project. For example, the files for the test source set are located in src/test/java.

Configure Gradle for test dependencies

To use the unit testing APIs, you need to configure the dependencies for your project. The default gradle build file for your project includes some of these dependencies by default, but you may need to add more dependencies for additional testing features such as matching or mocking frameworks.

In your app's top-level build.gradle file, specify these libraries as dependencies. Note that the version numbers for these libraries may have changed. If Android Studio reports a newer library, update the number to reflect the current version.

```
dependencies {
   // Required -- JUnit 4 framework
    testCompile 'junit:junit:4.12'
    // Optional -- hamcrest matchers
    testCompile 'org.hamcrest:hamcrest-library:1.3'
    // Optional -- Mockito framework
    testCompile 'org.mockito:mockito-core:1.10.19'
}
```

After you add dependencies to your build.gradle file you may have to sync your project to continue. Click **Sync Now** in Android Studio when prompted.

Configure a test runner

A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log. Your Android project has access to a basic JUnit test runner as part of the JUnit4 APIs. The Android test support library includes a test runner for instrumented and Espresso tests, `AndroidJUnitRunner`, which also supports Junit 3 and 4.

This chapter only demonstrates the default runner for unit tests. To set AndroidJUnitRunner as the default test runner in your Gradle project, add the following dependency to your build.gradle file. There may already be dependencies in the defaultConfig section. If so, add the testInstrumentationRunner line to that section.

```
android {
    defaultConfig {
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

Creating and running unit tests

Create your unit tests as a generic Java file using the JUnit 4 APIs, and store those tests in the **test** source set. Each Android Studio project template includes this source set and a sample Java test file called ExampleUnitTest.

Create a new test class

To create a new test class file, add a Java file to the test source set for your project. Test class files for unit testing are typically named for the class in your app that you are testing, with "Test" appended. For example, if your have a class called Calculator in your app, the class for your unit tests would be CalculatorTest.

To add a new test class file, use these steps:

1. Expand the java folder and the folder for your app's test source set. The existing unit test class files are shown.
2. Right-click on the test source set folder and select **New > Java Class.**
3. Name the file and click **OK**.

Write your tests

Use JUnit 4 syntax and annotations to write your tests. For example, the test class shown below includes the following annotations:

- The @RunWith annotation indicates the test runner that should be used for the tests in this class.
- The @SmallTest annotation indicates that this is a small (and fast) test.
- The @Before annotation marks a method as being the set up for the test.
- The @Test annotation marks a method as an actual test.

For more information on JUnit Annotations, see the JUnit Reference documentation.

```
@RunWith(JUnit4.class)
@SmallTest
public class CalculatorTest {
   private Calculator mCalculator;
   // Set up the environment for testing
   @Before
   public void setUp() {
      mCalculator = new Calculator();
   }

   // test for simple addition
   @Test
   public void addTwoNumbers() {
       double resultAdd = mCalculator.add(1d, 1d);
       assertThat(resultAdd, is(equalTo(2d)));
   }
}
```

The addTwoNumbers() method is the only actual test. The key part of a unit test is the assertion, which is defined here by the assertThat() method. Assertions are expressions that must evaluate and result in a value of true for the test to pass. JUnit 4 provides a number of assertion methods, but assertThat() is the most flexible, as it allows for general-purpose comparison methods called *matchers*. The Hamcrest framework is commonly used for matchers ("Hamcrest" is an anagram for matchers). Hamcrest includes a large number of comparison methods as well as enabling you to write your own.

For more information on assertions, see the JUnit reference documentation for the Assert class. For more information on the hamcrest framework, see the Hamcrest Tutorial.

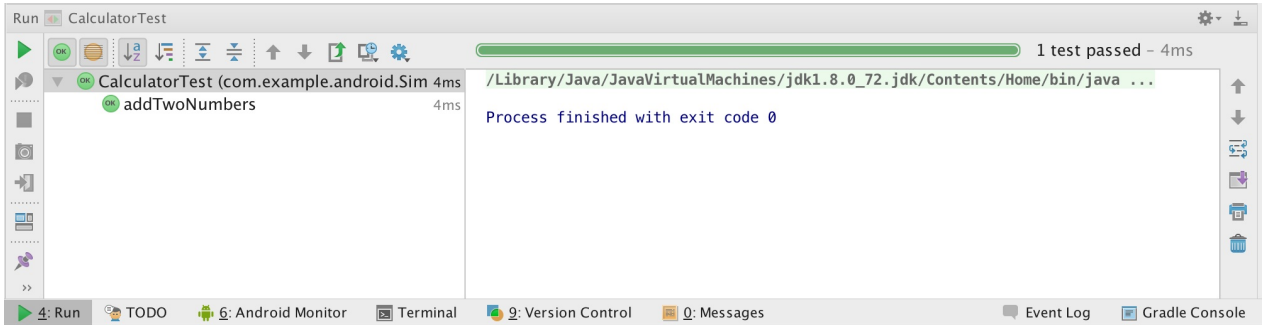
Note that the addTwoNumbers() method in this example includes only one assertion. The general rule for unit tests is to provide a separate test method for every individual assertion. Grouping more than one assertion into a single method can make your tests harder to debug if only one assertion fails, and obscures the tests that do succeed.

Run your tests

To run your local unit tests, use these steps:

- To run a single test, right-click that test method and select **Run**.
- To test all the methods in a test class, right-click the test file in the project view and select **Run**.
- To run all tests in a directory, right-click on the directory and select **Run tests**.

The project builds, if necessary, and the testing view appears at the bottom of the screen. If all the tests you ran are successful, the progress bar at the top of the view turns green. A status message in the footer also reports "Tests Passed."



Related Practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Testing Apps With Unit Tests

Learn More

- Best Practices for Testing
- Getting Started with Testing
- Building Local Unit Tests
- JUnit 4 Home Page
- JUnit 4 API Reference
- Mockito Home Page
- Android Testing Support - Testing Patterns (video)
- Android Testing Codelab
- Android Tools Protip: Test Size Annotations
- The Benefits of Using assertThat over other Assert Methods in Unit Tests

3.3: The Android Support Library

Contents:

- Introduction
- About the Android Support Library
- Support libraries and features
- Setting up and using the Android Support Library
- Related Practical
- Learn More

In this chapter you'll explore the Android Support Library, part of the Android SDK tools. You can use the Android Support Library for backward-compatible versions of new Android features and for additional UI elements and features not included in the standard Android framework.

About the Android Support Library

The Android SDK tools include a number libraries collectively called the *Android Support Library*. This package of libraries provides several features that are not built into the standard Android framework, and provides backward compatibility for older devices. Include any of these libraries in your app to incorporate that library's functionality.

Note: The Android Support library is a different package from the Android **Testing** Support library you learned about in a previous chapter. The testing support library provides tools and APIs just for testing, whereas the more general support library provides features of all kinds (but no testing).

Features

The features of the Android Support Library include:

- Backward-compatible versions of framework components. These compatibility libraries allow you to use features and components available on newer versions of the Android platform even when your app is running on an older platform version. For example, older devices may not have access to newer features such as fragments, action bars, or Material Design elements. The support library provides access to those features on older devices.
- Additional layout and user interface elements. The support library includes views and layouts that can be useful for your app, but are not included in the standard Android framework. For example, the RecyclerView view that you will use in a later chapter is part of the support library.
- Support for different device form factors, such as TV or wearables: For example, the Leanback library includes components specific to app development on TV devices.
- Design support: The design support library includes components to support Material Design elements in your app, including floating action buttons (FAB). You'll learn more about Material Design in a later chapter.
- Various other features such as palette support, annotations, percentage-based layout dimensions, and preferences.

Backward Compatibility

Support libraries allow apps running on older versions of the Android platform to support features made available on newer versions of the platform. For example, an app running on a version of Android lower than 5.0 (API level 21) that relies on framework classes cannot display Material Design elements, as that version of the Android framework doesn't support Material Design. However, if the app incorporates the v7 appcompat library, that app has access to many of the features available in API level 21, including support for Material Design. As a result, your app can deliver a more consistent experience across a broader range of platform versions.

The support library APIs also provide a compatibility layer between different versions of the framework APIs. This compatibility layer transparently intercepts API calls and changes either the arguments passed, handles the operation itself, or redirects the operation. In the case of the support libraries, by using the compatibility layer's methods, you can ensure interoperability between older and newer Android releases. Each new release of Android adds new classes and methods, and possibly deprecates some older classes and methods. The support libraries include compatibility classes that can be used for backward compatibility.You can identify these classes by their name as thier names include "Compat" (such as ActivityCompat).

When an app calls one of the support class's methods, the behavior of that method depends on the underlying Android version. If the device includes the necessary framework functionality, the support library uses the framework. If the device is running an older version of Android, the support library makes an attempt to implement similar compatible behavior with the APIs it has available.

For most cases you do not need to write complex code that checks the version of Android and performs different operations based on that version. You can rely on the support library to do those checks and choose appropriate behavior.

When in doubt, choose a support library compatibility class over the framework class.

Versions

Each package in the support library has a version number in three parts (X.Y.Z) that corresponds to an Android API level, and to a particular revision of that library. For example, a support library version number of 22.3.4 is version 3.4 of the support library for API 22.

As a general rule, use the most recent version of the support library for the API your app is compiled and targeted for, or a newer version. For example, if your app targets API 25, use the version 25.X.X of the support library.

You can always use a newer support library than the one for your targeted API. For example, if your app targets API 22 you can use version 25 or higher of the support library. The reverse is not true—you cannot use an older support library with a newer API. As a general rule, you should try to use the most up-to-date API and support libraries in your app.

API Levels

In addition to the actual version number, the name of the support library itself indicates the API level that the library is backward-compatible with. You cannot use a support library in your app for an API higher than the minimum API your app supports. For example, if the minimum API your app supports is 10, you cannot use the v13 support library or v14 preferences support library in your app. If your app uses multiple support libraries, your minimum API must be higher than the largest number -- that is, if you include support libraries for v7, v13, and v14 your minimum API must be at least 14.

All of the support libraries, including the v4 and v7 libraries, require a minimum SDK of API 9.

Support libraries and features

This section describes the important features provided by the libraries in the Android Support Library. You'll learn about many of the features described in this section in a later chapter.

v4 support library

The v4 support libraries include the largest set of APIs compared to the other libraries, including support for application components, user interface features, accessibility, data handling, network connectivity, and programming utilities.

The v4 support libraries include these specific components:

- v4 compat library: Compatibility wrappers (classes that include the word "Compat") for a number of core framework APIs.
- v4 core-utils library: Provides a number of utility classes
- v4 core-ui library: Implements a variety of UI-related components.

- v4 media-compat library: Backports portions of the media framework from API 21.
- v4 fragment library: Adds support for Android fragments.

v7 support library

The v7 support library includes both compatibility libraries and additional features.

The v7 support library includes all the v4 support libraries, so you don't have to add those separately. A dependency on the v7 support library is included in every new Android Studio project, and new activities in your project extend from AppCompatActivity.

The v7 support libraries include these specific components:

- v7 appcompat library: Adds support for the Action Bar user interface design pattern and support for material design user interface implementations.
- v7 cardview library: Provides the CardView class, a view that lets you show information inside cards.
- v7 gridlayout library: Includes the GridLayout class, which allows you to arrange user interface elements using a grid of rectangular cells
- v7 mediarouter library: Provides MediaRouter and related media classes that support Google Cast.
- v7 palette library: Implements the Palette class, which lets you extract prominent colors from an image.
- v7 recyclerview library: Provides the RecyclerView class, a view for efficiently displaying large data sets by providing a limited window of data items.
- v7 preference library: Provides APIs to support preference objects in app settings.

Other libraries

- v8 renderscript library: Adds support for the RenderScript, a framework for running computationally intensive tasks at high performance.
- v13 support library: Provides support for fragments with the FragmentCompat class and additional fragment support classes.
- v14 preference support library, and v17 preference support library for TV: provides APIs to add support for preference interfaces on mobile devices and TV.
- v17 leanback library: Provides APIs to support building user interfaces on TV devices.
- Annotations support library: Contains APIs to support adding annotation metadata to your apps.
- Design support library: Adds support for various Material Design components and patterns such as navigation drawers, floating action buttons (FAB), snackbars, and tabs.
- Custom Tabs support library: Adds support for adding and managing custom tabs in your apps.
- Percent support library: Enables you to add and manage percentage based dimensions in your app.
- App recommendation support library for TV: Provides APIs to support adding content recommendations in your app running on TV devices.

Setting up and using the Android Support Library

The Android Support Library package is part of the Android SDK, and available to download in the Android SDK manager. To set up your project to use any of the support libraries, use these steps:

1. Download the support library with the Android SDK manager, or verify that the support libraries are already available.
2. Find the library dependency statement for the support library you're interested in.
3. Add that dependency statement to your build.gradle file.

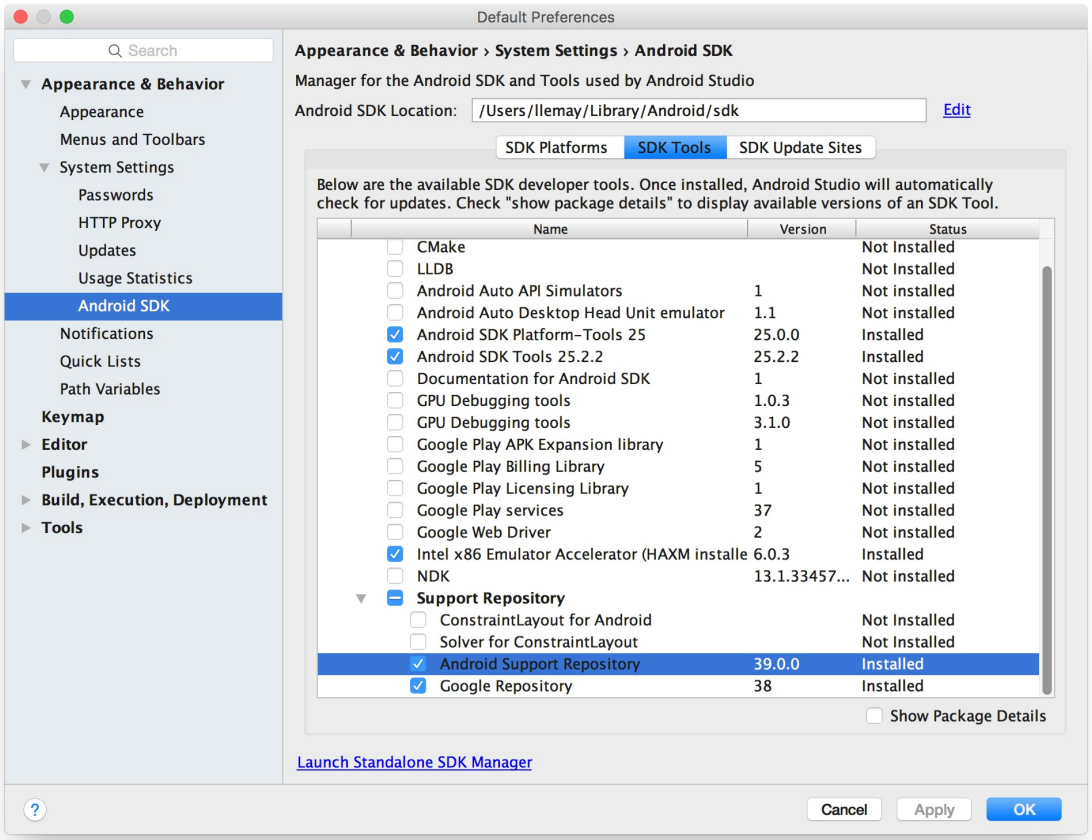
Download the support library

In Android Studio, you'll use the Android Support Repository—the repository in the SDK manager for all support libraries—to get access to the library from within your project.

You may already have the Android support libraries downloaded and installed with Android Studio. To verify that you have the support libraries available, follow these steps:

1. In Android Studio, select **Tools > Android > SDK Manager**, or click the SDK Manager icon.

   The SDK Manager preference pane appears.



2. Click the **SDK Tools** tab and expand Support Repository.
3. Look for **Android Support Repository** in the list.
   - If **Installed** appears in the Status column, you're all set. Click **Cancel**.
   - If **Not installed** or **Update Available** appears, click the checkbox next to Android Support Repository. A download icon should appear next to the checkbox. Click **OK.**
4. Click **OK** again, and then **Finish** when the support repository has been installed.

Find a library dependency statement

To provide access to a support library from your project, you add that library to your gradle build file as a dependency. Dependency statements have a specific format that includes the name and version number of the library.

1. Visit the Support Library Features page on developer.android.com.
2. Find the library you're interested in on that page, for example, the Design Support Library for Material Design support.
3. Copy the dependency statement shown at the end of the section. For example, the dependency for the design support library looks like this:

```
com.android.support:design:23.3.0
```

   The version number at the end of the line may vary from the one shown above. You will update the version number when you add the dependency to the build.gradle file in the next step.

Add the dependency to your build.gradle file

The gradle scripts for your project manage how your app is built, including specifying the dependencies your app has on other libraries. To add a support library to your project, modify your gradle build files to include the dependency to that library you found in the previous section.

1. In Android Studio, make sure the **Project** pane is open and the Android tab is clicked.
2. Expand **Gradle Scripts**, if necessary, and open the **build.gradle (Module: app)** file.

   Note that build.gradle for the overall project (build.gradle (Project: app_name) is a different file from the build.gradle for the app module.

3. Locate the `dependencies` section of build.gradle, near the end of the file.

   The dependencies section for a new project may already include dependencies several other libraries.

4. Add a dependency for the support library that includes the statement you copied in the previous task. For example, a complete dependency on the design support library looks like this:

   ```
   compile 'com.android.support:design:23.3.0'
   ```

5. Update the version number, if necessary.

   If the version number you specified is lower than the currently available library version number, Android Studio will warn you that an updated version is available. ("a newer version of `com.android.support:design` is available"). Edit the version number to the updated version, or type Shift+Enter and choose "Change to XX.X.X" where XX.X.X is the updated version number.

6. Click **Sync Now** to sync your updated gradle files with the project, if prompted.

Using the support library APIs

All the support library classes are contained in the android.support packages, for example, android.support.v7.app.AppCompatActivity is the fully-qualified name for the AppCompatActivity class, from which all of your activities extend.

Support Library classes that provide support for existing framework APIs typically have the same name as framework class but are located in the android.support class packages. Make sure that when you import those classes you use the right package name for the class you're interested in. For example, when applying the ActionBar class, use one of:

- `android.support.v7.app.ActionBar` when using the Support Library.
- `android.app.ActionBar` when developing only for API level 11 or higher.

The support library also includes several View classes used in XML layout files. In the case of the views, you must always use the fully-qualified name of that view in the XML element for that view:

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
</android.support.design.widget.CoordinatorLayout>
```

Note: You'll learn about CoordinatorLayout in a later chapter.

Checking system versions

Although the support library can help you implement single apps that work across Android platform versions, there may be times when you need to check for the version of Android your app is running on, and provide the correct code for that version.

Android provides a unique code for each platform version in the Build constants class. Use these codes within your app to test for the version and to ensure that the code that depends on higher API levels is executed only when those APIs are available on the system.

```
private void setUpActionBar() {
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        ActionBar actionBar = getActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    } else { // do something else }
}
```

Related Practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Using The Android Support Libraries

Learn More

- Android Support Library (introduction)
- Support Library Setup
- Support Library Features
- Supporting Different Platform Versions
- Picking your compileSdkVersion, minSdkVersion, and targetSdkVersion
- All the Things Compat
- API Reference (all packages that start with android.support)

4.1: User Input Controls

- Interaction design for user input
- Input controls and view focus
- Using buttons
- Using input controls for making choices
- Text input
- Using dialogs and pickers
- Recognizing gestures
- Related practical
- Learn more

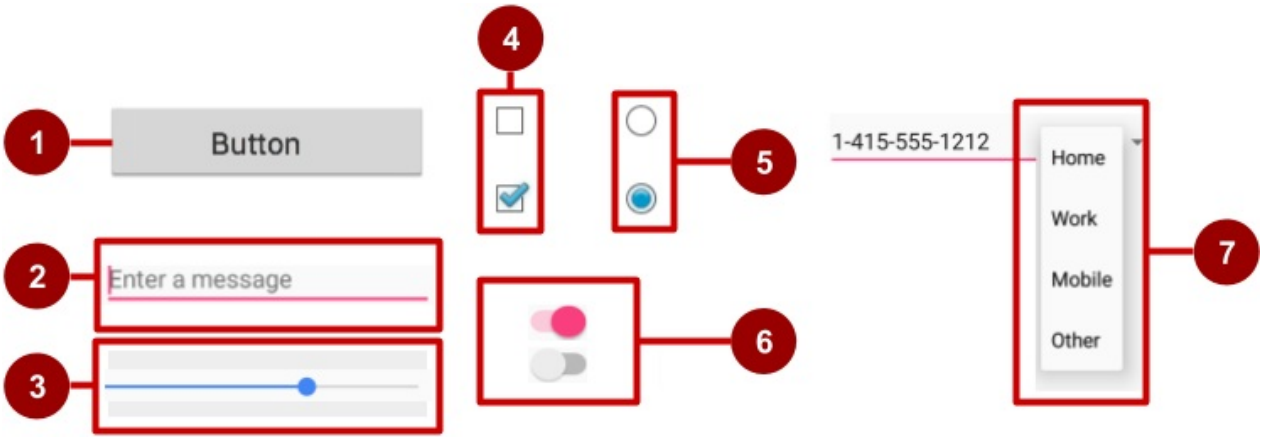
Interaction design for user input

The reason you design an app is to provide some function to a user, and in order to use it, there must be a way for the user to *interact* with it. For an Android app, interaction typically includes tapping, pressing, typing, or talking and listening. And the framework provides corresponding user interface (UI) elements such as buttons, menus, keyboards, text entry fields, and a microphone.

In this chapter you will learn how to design an app's user interaction—the buttons needed for triggering actions, and the text entry fields for user input. In your design you need to anticipate what users might need to do, and ensure that the UI has elements that are easy to access, understand, and use. When your app needs to get data from the user, make it easy and obvious. Give the user whatever assistance possible to provide any input data, such as anticipating the source of the data, minimizing the number of user gestures (such as taps and swipes), and pre-filling forms if possible.

Make sure your app is intuitive; that is, your app should perform as your users expect it to perform. When you rent a car, you expect the steering wheel, gear shift, headlights, and indicators to be in a certain place. When you enter a room, you expect the light switch to be in a certain place. When a user starts an app, the user expects buttons to be clickable, spinners to show a drop-down menu, and text editing fields to show the onscreen keyboard when tapping inside them. Don't violate the established expectations, or you'll make it harder for your users to use your app.

Note: Android users have become familiar with UI elements acting in a certain way, so it is important to be consistent with the experience of other Android apps, and predictable in your choices and their layout. Doing so helps you make apps that satisfy your customers.
This chapter introduces the Android *input controls,* which are the interactive components in your app's user interface. You can use a wide variety of input controls in your UI, such as text fields, buttons, checkboxes, radio buttons, toggle buttons, spinners, and more.



In the above figure:

1. Button

2. Text field
3. Seek bar
4. Checkboxes
5. Radio buttons
6. Toggle
7. Spinner

For a brief description of each input control, see Input Controls in the developer documentation.

Input controls and view focus

Android applies a common programmatic abstraction to all input controls called a *view*. The View class represents the basic building block for UI components, including input controls. In previous chapters, we have learned that `View` is the base class for classes that provide support for interactive UI components, such as buttons, text fields, and layout managers.

If there are many UI input components in your app, which one gets input from the user first? For example, if you have several TextView objects and an EditText object in your app, which UI component (that is, which View) receives text typed by the user first?

The View that "has the focus" will be the component that receives user input.

Focus indicates which view is currently selected to receive input. Focus can be initiated by the user by touching a View, such as a TextView or an EditText object. You can define a focus order in which the user is guided from UI control to UI control using the Return key, Tab key, or arrow keys. Focus can also be programmatically controlled; a programmer can `requestFocus()` on any View that is focusable.

Another attribute of an input control is *clickable*. If this attribute is (boolean) `true`, then the View can react to click events. As it is with focus, clickable can be programmatically controlled.

The difference between *clickable* and *focusable* is that clickable means the view can be clicked or tapped, while focusable means that the view is allowed to gain focus from an input device such as a keyboard. Input devices like keyboards can't determine which view to send their input events to, so they send them to the view that has focus.

Android device input methods are becoming quite diverse: directional pads, trackballs, touch screens, keyboards, and more. Some devices, like tablets and smartphones, are primarily navigated by touch. Others, like the Google TV, have no touch screen whatsoever and rely upon input devices such as those with a directional pad (d-pad). When a user is navigating through a user interface with an input device such as directional keys or a trackball, it is necessary to:

- Make it visually clear which view has focus, so that the user knows where the input goes.
- Explicitly set the focus in your code to provide a path for users to navigate through the input elements using directional keys or a trackball.

Fortunately, in most cases you don't need to control focus yourself, unless you want to provide a set of text input fields and you want the user to be able to move from one field to the next by tapping the Return or Tab key. Android provides "touch mode" for devices that can be touched, such as smartphones and tablets. When the user begins interacting with the interface by touching it, only Views with `isFocusableInTouchMode()` set to `true` are focusable, such as text input fields. Other Views that are touchable, such as buttons, do not take focus when touched. If the user hits a directional key or scrolls with a trackball, the device exits "touch mode" and finds a view to take focus.

Focus movement is based on an algorithm that finds the nearest neighbor in a given direction:

- When the user touches the screen, the topmost view under the touch is in focus, providing touch-access for the child views of the topmost view.
- If you set an EditText view to a single-line, the user can tap the Return key on the keyboard to close the keyboard and shift focus to the next input control view based on what the Android system finds:
  - The system usually finds the nearest input control in the same direction the user was navigating (up, down, left, or right).
  - If there are multiple input controls that are nearby and in the same direction, the system scans from left to right,

top to bottom.
- Focus can also shift to a different view if the user interacts with a directional control, such as a directional pad (d-pad) or trackball.

You can influence the way Android handles focus by arranging input controls such as EditText elements in a certain layout from left to right and top to bottom, so that focus shifts from one to the other in the sequence you want.

If the algorithm does not give you what you want, you can override it by adding the `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, and `nextFocusUp` XML attributes to your layout file.

1. Add one of these attributes to a view to decide where to go upon leaving the view—in other words, which view should be the *next* view.
2. Define the value of the attribute to be the `id` of the next view. For example:

```
<LinearLayout
android:orientation="vertical"
... >
<Button android:id="@+id/top"
      android:nextFocusUp="@+id/bottom"
      ... />
<Button android:id="@+id/bottom"
      android:nextFocusDown="@+id/top"
      ... />
</LinearLayout>
```

Ordinarily in a vertical `LinearLayout`, navigating up from the first `Button` would not go anywhere, nor would navigating down from the second `Button`. But in the above example, the top `Button` has defined the bottom `button` as the `nextFocusUp` (and vice versa), so the navigation focus will cycle from top-to-bottom and bottom-to-top.

If you'd like to declare a View as focusable in your UI (when it is traditionally not), add the `android:focusable` XML attribute to the View in the layout, and set its value to `true`. You can also declare a View as focusable while in "touch mode" with `android:focusableInTouchMode` set to `true`.

You can also explicitly set the focus or find out which view has focus by using the following methods:

- Call onFocusChanged to determine where focus came from.
- To find out which view currently has the focus, call Activity.getCurrentFocus(), or use ViewGroup.getFocusedChild() to return the focused child of a view (if any).
- To find the view in the hierarchy that currently has focus, use findFocus().
- Use requestFocus to give focus to a specific view.
- To change whether a view can take focus, call setFocusable.
- To set a listener that will be notified when the view gains or loses focus, use setOnFocusChangeListener.

Understanding focus with respect to input controls is essential for understanding how the on-screen keyboard works with text editing views. For example, depending on which attributes you use with an EditText view, tapping the Return key in the keyboard can either enter a new line, or advance the focus to the next view. You'll learn more about focus with text editing views later in this chapter.

Using buttons

People like to press buttons. Show someone a big red button with a message that says "Do not press" and the person will likely press it for the sheer pleasure of pressing a big red button (that the button is forbidden is also a factor).

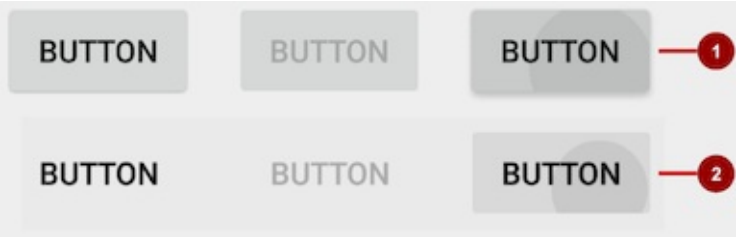
You can make a Button using:

- Only text, as shown on the left side of the figure below.
- Only an icon, as shown in the center of the figure below.
- Both text and an icon, as shown on the right side of the figure below.

When touched or clicked, a button performs an action. The text and/or icon provides a hint of that action. It is also referred to as a "push-button" in Android documentation.

A button is a rectangle or rounded rectangle, wider than it is tall, with a descriptive caption in its center. Android buttons follow the guidelines in the the Android Material Design Specification—you will learn more about that in a later lesson.

Android offers several types of buttons, including raised buttons and flat buttons as shown in the figure below. These buttons have three states: normal, disabled, and pressed.



In the above figure:

1. Raised button in three states: normal, disabled, and pressed.
2. Flat button in three states: normal, disabled, and pressed.

Designing raised buttons

A raised button is a rectangle or rounded rectangle that appears lifted from the screen—the shading around it indicates that it is possible to touch or click it. The raised button can show text or an icon, or show both text and an icon.

To use raised buttons that conform to the Material Design Specification, follow these steps:

1. In your **build.gradle (Module: app)** file, add the newest appcompat library to the `dependencies` section:

   ```
   compile 'com.android.support:appcompat-v7:x.x.x.'
   ```

   In the above, *x.x.x.* is the version number. If the version number you specified is lower than the currently available library version number, Android Studio will warn you ("a newer version is available"). Update the version number to the one Android Studio tells you to use.

2. Make your activity extend `android.support.v7.app.AppCompatActivity`:

   ```
   public class MainActivity extends AppCompatActivity {
   ...
   }
   ```

3. Use the `Button` element in the layout file. There is no need for an additional attribute, as a raised button is the default style.

   ```
   <Button
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
   ... />
   ```

Use raised buttons to give more prominence to actions in layouts with a lot of varying content. Raised buttons add dimension to a flat layout—they emphasize functions on busy or wide spaces. Raised buttons show a background shadow when touched (pressed) or clicked, as shown below.

In the above figure:

1. Normal state: In its normal state, the button looks like a raised button.
2. Disabled state: When the button is disabled, it is grayed out and it's not active in the app's context. In most cases you would hide an inactive button, but there may be times when you would want to show it as disabled.
3. Pressed state: The pressed state, with a larger background shadow, indicates that the button is being touched or clicked. When you attach a callback to the button (such as the OnClick attribute), the callback is called when the button is in this state.

Creating a raised button with text

Some raised buttons are best designed as text, without an icon, such as a "Save" button, because an icon by itself might not convey an obvious meaning. To create a raised button with text, use the Button class, which extends the TextView class.

To create a raised button with just text, use the `Button` class in your XML layout as follows:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    ... />
```

The best practice with text buttons is to define a very short word as a string resource (`button_text` in the above example), so that the string can be translated. For example, "Save" could be translated into French as "Enregistrer" without changing any of the code.

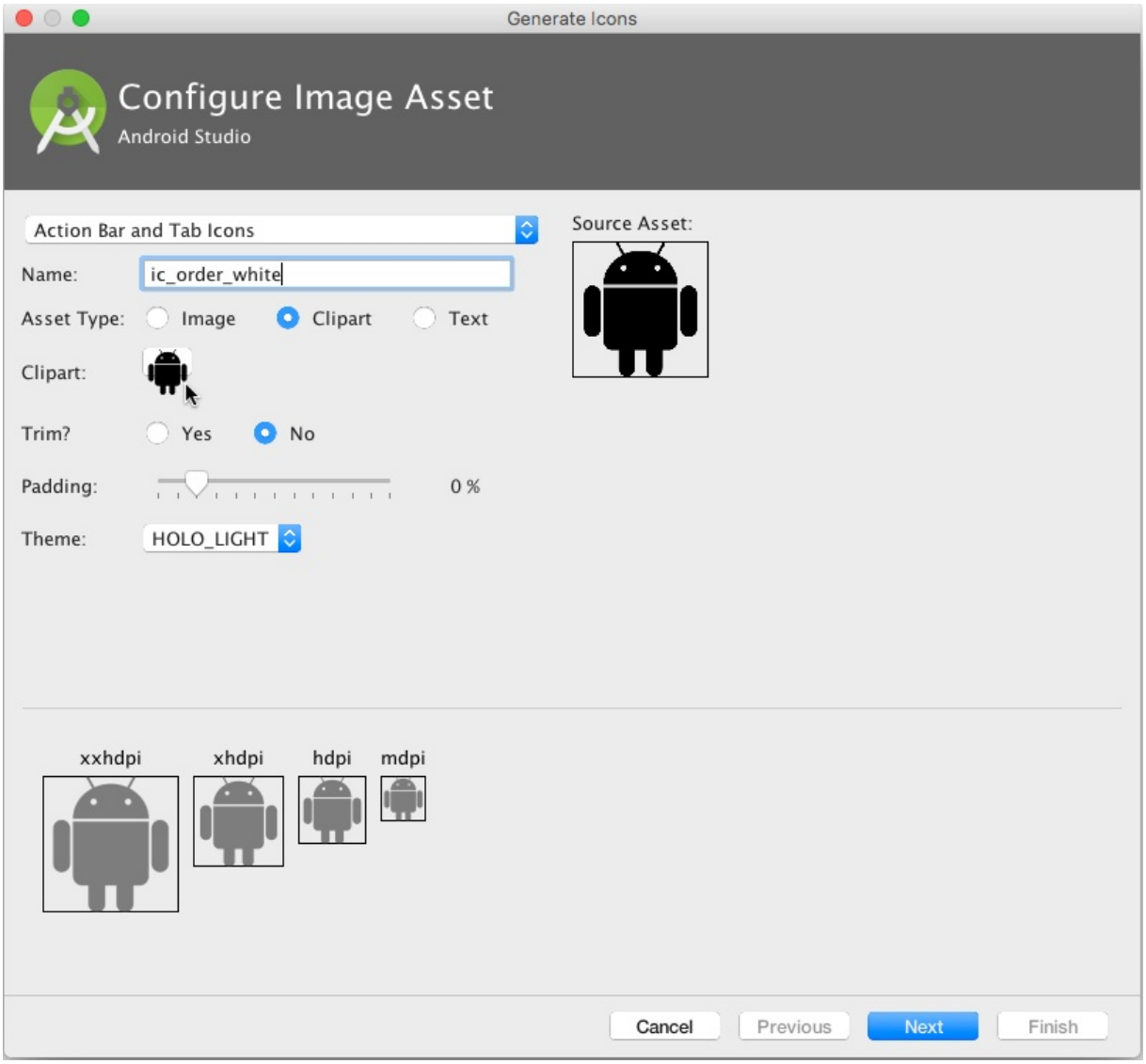
Creating a raised button with an icon and text

While a button usually displays text that tells the user what the button is for, raised buttons can also display icons along with text.

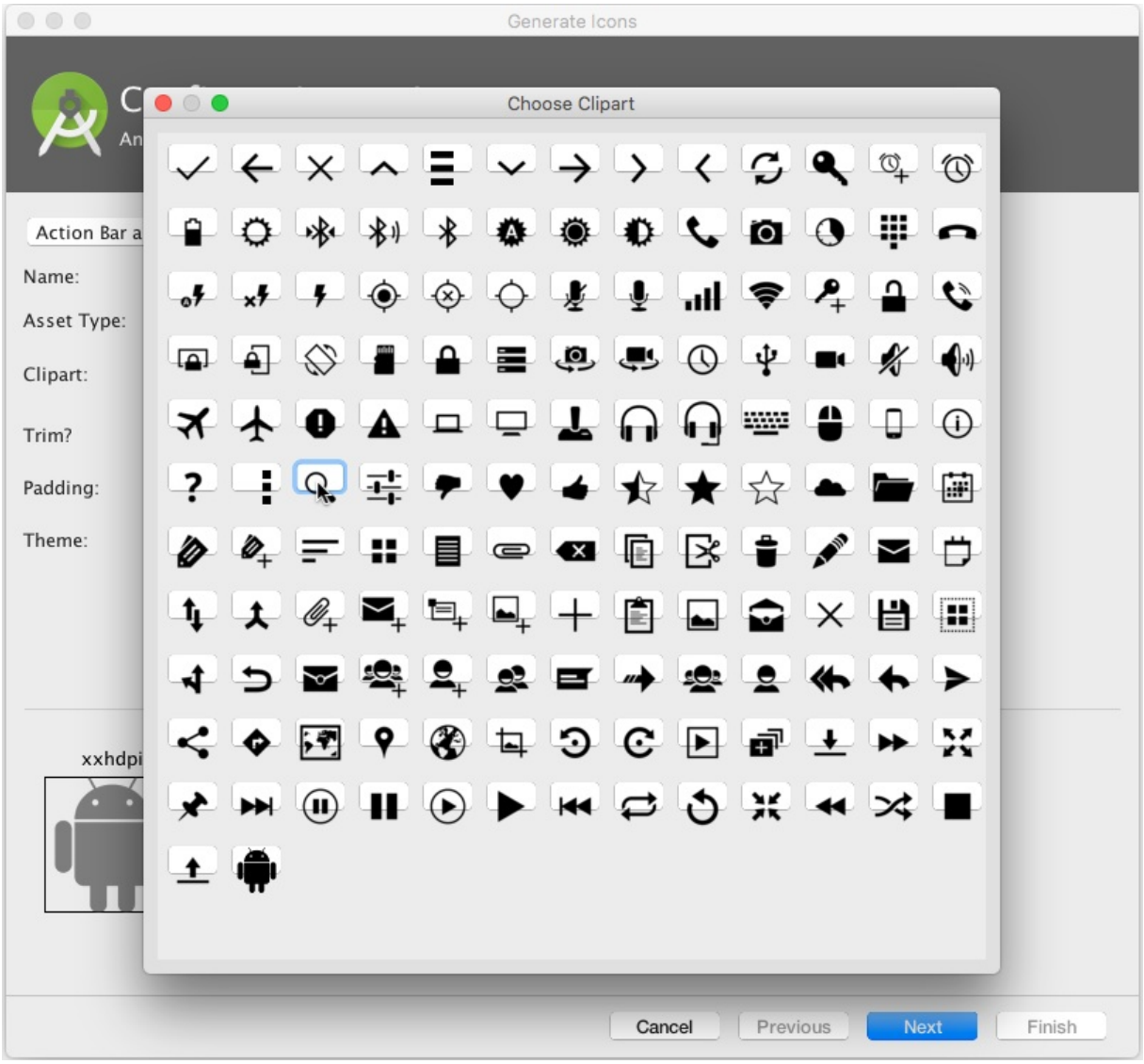
Choosing an icon

To choose images of a standard icon that are resized for different displays, follow these steps:

1. Expand **app > res** in the Project view, and right-click (or Command-click) **drawable**.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.



3. Choose **Action Bar and Tab Items** in the drop-down menu of the Configure Image Asset dialog (see Image Asset Studio for a complete description of this dialog.)
4. Click the **Clipart:** image (the Android logo) to select a clipart image as the icon. A page of icons appears as shown below. Click the icon you want to use.



5. You may want to make the following adjustments:
   - Choose **HOLO_DARK** from the Theme drop-down menu to sets the icon to be white against a dark-colored (or black) background.
   - Depending on the shape of the icon, you may want to add padding to the icon so that the icon doesn't crowd the text. Drag the Padding slider to the right to add more padding.
6. Click **Next**, and then click **Finish** in the Confirm Icon Path dialog. The icon name should now appear in the **app > res > drawable** folder.

Vector images of a standard icon are automatically resized for different sizes of device displays. To choose vector images, follow these steps:

1. Expand **app > res** in the Project view, and right-click (or Command-click) **drawable**.
2. Choose **New > Vector Asset** for an icon that automatically resizes itself for each display.
3. The Vector Asset Studio dialog appears for a vector asset. Click the **Material Icon** radio button, and then click the **Choose** button to choose an icon from the Material Design spec (see Add Multi-Density Vector Graphics for a complete description of this dialog).
4. Click **Next** after choosing an icon, and click **Finish** to finish. The icon name should now appear in the **app > res > drawable** folder.

Adding the button with text and icon to the layout

To create a button with text and an icon as shown in the figure below, use a `Button` in your XML layout. Add the `android:drawableLeft` attribute to draw the icon to the left of the button's text, as shown in the figure below:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    ... />
```

Creating a raised button with only an icon

If the icon is universally understood, you may want to use it instead of text.

To create a raised button with just an icon or image (no text), use the ImageButton class, which extends the ImageView class. You can add an `ImageButton` to your XML layout as follows:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    ... />
```

Changing the style and appearance of raised buttons

The simplest way to show a more prominent raised button is to use a different background color for the button. You can specify the `android:background` attribute with a drawable or color resource:

```
android:background="@color/colorPrimary"
```

The appearance of your button—the background color and font—may vary from one device to another, because devices by different manufacturers often have different default styles for input controls. You can control exactly how your buttons and other input controls are styled using a *theme* that you apply to your entire app.

For instance, to ensure that all devices that can run the Holo theme will use the Holo theme for your app, declare the following in the `<application>` element of the AndroidManifest.xml file:

```
android:theme="@android:style/Theme.Holo"
```

After adding the above declaration, the app will be displayed using the theme.

Apps designed for Android 4.0 and higher can also use the DeviceDefault public theme family. DeviceDefault themes are aliases for the device's native look and feel. The DeviceDefault theme family and widget style family offer ways for developers to target the device's native theme with all customizations intact.

For Android apps running on 4.0 and newer, you have the following options:

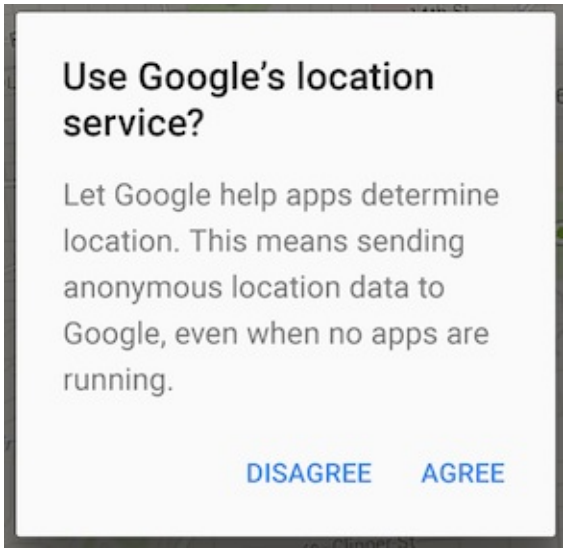
- Use a theme, such as one of the Holo themes, so that your app has the exact same look across all Android devices running 4.0 or newer. In this case, the app's look does not change when running on a device with a different default skin or custom skin.
- Use one of the DeviceDefault themes so that your app takes on the look of the device's default skin.
- Don't use a theme, but you may have unpredictable results on some devices.

If you're not already familiar with Android's style and theme system, you should read Styles and Themes. The blog post "Holo Everywhere" provides information about using the Holo theme while supporting older devices.

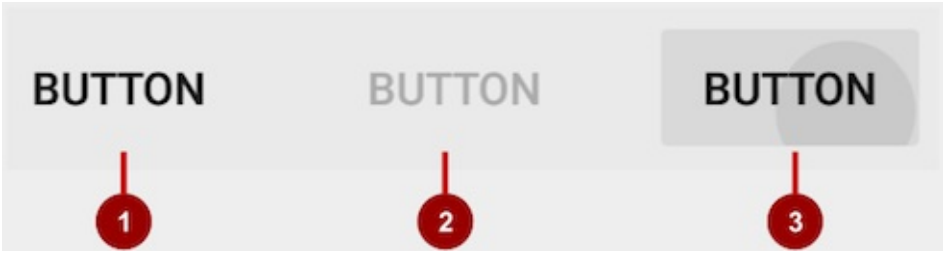
For a guide on styling and customizing buttons using XML, see Buttons (in the "User Interface" section of the Android developer guide). For a comprehensive guide to designing buttons, see "Components - Buttons" in the Material Design Specification.

Designing flat buttons

A flat button, also known as a borderless button, is a text-only button that appears flat on the screen without a shadow. The major benefit of flat buttons is simplicity — they minimize distraction from content. Flat buttons are useful when you have a dialog, as shown in the figure below, which requires user input or interaction. In this case, you would want to have the same font and style as the text surrounding the button. This keeps the look and feel the same across all elements within the dialog.


Use Google's location service?
Let Google help apps determine location. This means sending anonymous location data to Google, even when no apps are running.
DISAGREE
AGREE

Flat buttons, shown below, resemble basic buttons except that they have no borders or background, but still change appearance during different states. A flat button shows an ink shade around it when pressed (touched or clicked).



In the above figure:

1. Normal state: In its normal state, the button looks just like ordinary text.
2. Disabled state: When the text is grayed out, the button is not active in the app's context.
3. Pressed state: The pressed state, with a background shadow, indicates that the button is being touched or clicked. When you attach a callback to the button (such as the `android:onClick` attribute), the callback is called when the button is in this state.
   Note: If you use a flat button within a layout, be sure to use padding to set it off from the surrounding text, so that the user can easily see it.

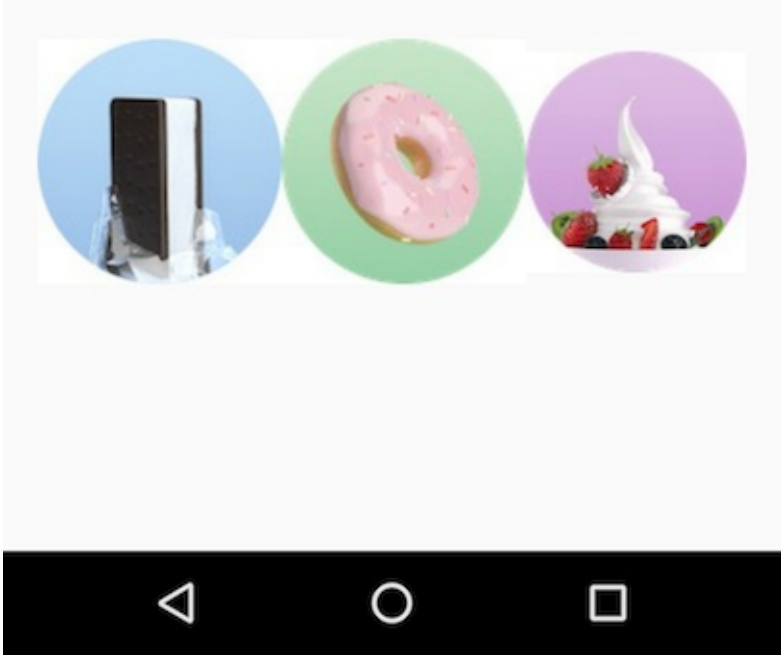
To create a flat button, use the Button class. Add a `Button` to your XML layout, and apply `"?android:attr/borderlessButtonStyle"` as the `style` attribute:

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage"
    style="?android:attr/borderlessButtonStyle" />
```

Designing images as buttons

You can turn any View, such as an ImageView, into a button by adding the `android:onClick` attribute in the XML layout. The image for the ImageView must already be stored in the **drawables** folder of your project.

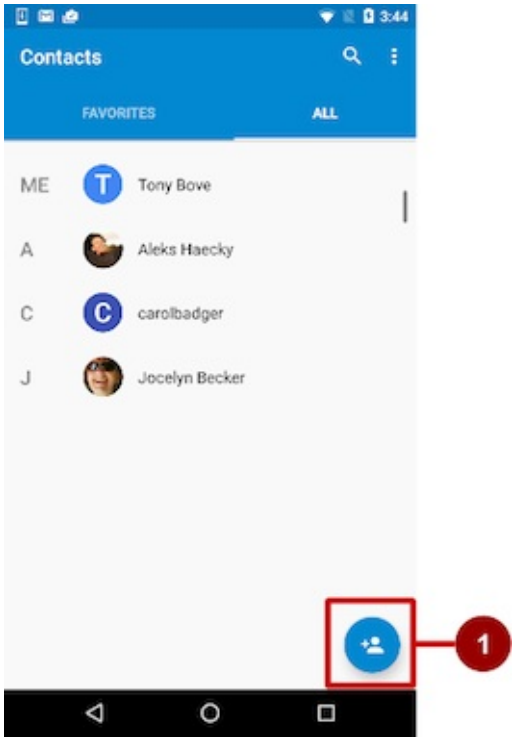
Note: To bring images into your Android Studio project, create or save the image in JPEG format, and copy the image file into the **app > src > main > res > drawables** folder of your project. For more information about drawable resources, see Drawable Resources in the App Resources section of the Android Developer Guide.
If you are using multiple images as buttons, arrange them in a viewgroup so that they are grouped together. For example, the following images in the drawable folder (icecream_circle.jpg, donut_circle.jpg, and froyo_circle.jpg) are defined for ImageViews that are grouped in a LinearLayout set to a horizontal orientation so that they appear side-by-side:



```
<LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal"
        android:layout_marginTop="260dp">
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/icecream_circle"
            android:onClick="orderIcecream"/>
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/donut_circle"
            android:onClick="orderDonut"/>
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/froyo_circle"
            android:onClick="orderFroyo"/>
    </LinearLayout>
```

Designing a floating action button

A floating action button, shown below as #1 in the figure below, is a circular button that appears to float above the layout.



You should use a floating action button only to represent the primary action for a screen. For example, the primary action for the Contacts app's main screen is adding a contact, as shown in the figure above. A floating action button is the right choice if your app requires an action to be persistent and readily available on a screen. Only one floating action button is recommended per screen.

The floating action button uses the same type of icons that you would use for a button with an icon, or for actions in the app bar at the top of the screen. You can add an icon as described previously in "Choosing an icon for the button".

To use a floating action button in your Android Studio project, you must add the following statement to your **build.gradle (Module: app)** file in the `dependencies` section:

```
compile 'com.android.support:design:23.4.0'
```

Note: The version number at the end of the statement may change; use the newest version suggested by Android Studio. To create a floating action button, use the FloatingActionButton class, which extends the ImageButton class. You can add a floating action button to your XML layout as follows:

```
<android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@drawable/ic_fab_chat_button_white" />
```

Floating action buttons, by default, are 56 x 56 dp in size. It is best to use the default size unless you need the smaller version to create visual continuity with other screen elements.

You can set the *mini* size (30 x 40 dp) with the `app:fabSize` attribute:

```
app:fabSize="mini"
```

To set it back to the default size (56 x 56 dp):

```
app:fabSize="normal"
```

For more design instructions involving floating action buttons, see Components– Buttons: Floating Action Button in the Material Design Spec.

Responding to button-click events

Use an *event listener* called OnClickListener, which is an interface in the View class, to respond to the click event that occurs when the user taps or clicks a clickable object, such as a Button, ImageButton, or FloatingActionButton. For more information on event listeners, or other types of UI events, read the Input Events section of the Android Developer Documentation.

Adding onClick to the layout element

To set up an OnClickListener for the clickable object in your Activity code and assign a callback method, use the `android:onClick` attribute with the clickable object's element in the XML layout. For example:

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

In this case, when a user clicks the button, the Android system calls the Activity's `sendMessage()` method:

```
public void sendMessage(View view) {
    // Do something in response to button click
}
```

The method you declare as the `android:onClick` attribute must be `public`, return `void`, and define a `View` as its only parameter (this will be the view that was clicked). Use the method to perform a task or call other methods as a response to the button click.

Using the button-listener design pattern

You can also handle the click event programmatically using the button-listener design pattern (see figure below). For more information on the "listener" design pattern, see Creating Custom Listeners.

Use the event listener View.OnClickListener, which is an interface in the View class that contains a single callback method, onClick(). The method is called by the Android framework when the view is triggered by user interaction.

The event listener must already be registered to the view in order to be called for the event. Follow these steps to register the listener and use it (refer to the figure below the steps):

1. Use the findViewById() method of the View class to find the button in the XML layout file:

   ```
   Button button = (Button) findViewById(R.id.button_send);
   ```

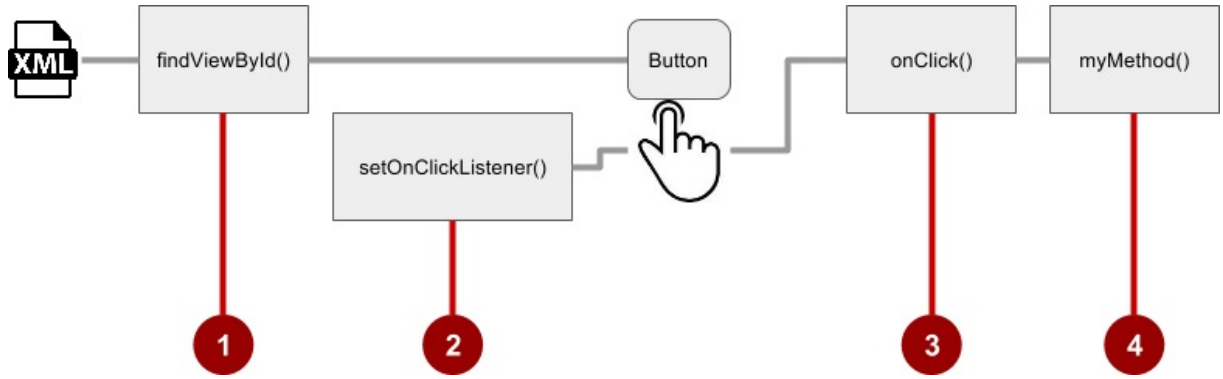
2. Get a new `View.OnClickListener` object and register it to the button by calling the setOnClickListener() method. The argument to `setOnClickListener()` takes an object that implements the View.OnClickListener interface, which has one method: `onClick()`.

   ```
   button.setOnClickListener(new View.OnClickListener() {
   ...
   ```

3. Define the `onClick()` method to be `public`, return `void`, and define a `View` as its only parameter:

```
public void onClick(View v) {
    // Do something in response to button click
}
```

4. Create a method to do something in response to the button click, such as perform an action.



To set the click listener programmatically instead of with the onClick attribute, customize the View.OnClickListener class and override its `onClick()` handler to perform some action, as shown below:

```
fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // Add a new word to the wordList.
            }
        });
```

Using the event listener interface for other events

Other events can occur with UI elements, and you can use the callback methods already defined in the event listener interfaces to handle them. The methods are called by the Android framework when the view—to which the listener has been registered—is triggered by user interaction. You therefore must set the appropriate listener to use the method. The following are some of the listeners available in the Android framework and the callback methods associated with each one:

- `onClick()` from View.OnClickListener: Handles a click event in which the user touches and then releases an area of the device display occupied by a view. The onClick() callback has no return value.
- `onLongClick()` from View.OnLongClickListener: Handles an event in which the user maintains the touch over a view for an extended period. This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return `true` to indicate that you have handled the event and it should stop here; return `false` if you have not handled it and/or the event should continue to any other on-click listeners.
- `onTouch()` from View.OnTouchListener: Handles any form of touch contact with the screen including individual or multiple touches and gesture motions, including a press, a release, or any movement gesture on the screen (within the bounds of the UI element). A MotionEvent is passed as an argument, which includes directional information, and it returns a boolean to indicate whether your listener consumes this event.
- `onFocusChange()` from View.OnFocusChangeListener: Handles when focus moves away from the current view as the result of interaction with a trackball or navigation key.
- `onKey()` from View.OnKeyListener: Handles when a key on a hardware device is pressed while a view has focus.

Using input controls for making choices

Android offers ready-made input controls for the user to select one or more choices:

- Checkboxes: Select one or more values from a set of values by clicking each value's checkbox.
- Radio buttons: Select only one value from a set of values by clicking the value's circular "radio" button. If you are providing only two or three choices, you might want to use radio buttons for the choices if you have room in your layout

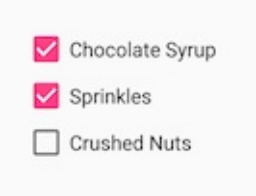
for them.

- Toggle button: Select one state out of two or more states. Toggle buttons usually offer two visible states, such as "on" and "off".
- Spinner: Select one value from a set of values in a drop-down menu. Only one value can be selected. Spinners are useful for three or more choices, and takes up little room in your layout.

Checkboxes

Use checkboxes when you have a list of options and the user may select *any number* of choices, including no choices. Each checkbox is independent of the other checkboxes in the list, so checking one box doesn't uncheck the others. (If you want to limit the user's selection to only one item of a set, use radio buttons.) A user can also uncheck an already checked checkbox.

Users expect checkboxes to appear in a vertical list, like a to-do list, or side-by-side horizontally if the labels are short.



Each checkbox is a separate instance of the CheckBox class. You create each checkbox using a `CheckBox` element in your XML layout. To create multiple checkboxes in a vertical orientation, use a vertical LinearLayout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <CheckBox android:id="@+id/checkbox1_chocolate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/chocolate_syrup" />
        <CheckBox android:id="@+id/checkbox2_sprinkles"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/sprinkles" />
        <CheckBox android:id="@+id/checkbox3_nuts"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/crushed_nuts" />

</LinearLayout>
```

Typically programs retrieve the state of checkboxes when a user touches or clicks a **Submit** or **Done** button in the same activity, which uses the `android:onClick` attribute to call a method such as `onSubmit()` :

```
<Button
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="@string/submit"
   android:onClick="onSubmit"/>
```

The callback method— `onSubmit()` in the above `Button` example—must be `public` , return `void` , and define a `View` as a parameter (the view that was clicked). In this method you can determine if a checkbox is selected by using the isChecked() method (inherited from CompoundButton). The `isChecked()` method will return a (boolean) `true` if there is a checkmark in the box. For example, the following statement assigns the boolean value of `true` or `false` to `checked` depending on whether the checkbox is checked:

```
boolean checked = ((CheckBox) view).isChecked();
```

The following code snippet shows how the `onSubmit()` method might check to see which checkbox is selected, using the resource `id` for the checkbox element:

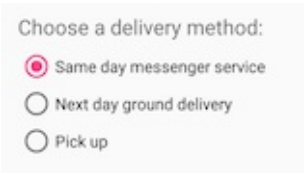
```
public void onSubmit(View view){
   StringBuffer toppings = new
                  StringBuffer().append(getString(R.string.toppings_label));
   if  (((CheckBox) findViewById(R.id.checkbox1_chocolate)).isChecked()) {
      toppings.append(getString(R.string.chocolate_syrup_text));
   }
   if  (((CheckBox) findViewById(R.id.checkbox2_sprinkles)).isChecked()) {
      toppings.append(getString(R.string.sprinkles_text));
   }
   if  (((CheckBox) findViewById(R.id.checkbox3_nuts)).isChecked()) {
      toppings.append(getString(R.string.crushed_nuts_text));
   }
...
}
```

Tip: To respond quickly to a checkbox—such as display a message (like an alert), or show a set of further options—you can use the `android:onClick` attribute in the XML layout for each checkbox to declare the callback method for that checkbox, which must be defined within the activity that hosts this layout.

For more information about checkboxes, see Checkboxes in the User Interface section of the Android Developer Documentation.

Radio buttons

Use radio buttons when you have two or more options that are mutually exclusive—the user must select only one of them.



(If you want to enable more than one selection from the set, use checkboxes.)

Users expect radio buttons to appear as a vertical list, or side-by-side horizontally if the labels are short.

Each radio button is an instance of the RadioButton class. Radio buttons are normally used together in a RadioGroup. When several radio buttons live inside a radio group, checking one radio button unchecks all the others. You create each radio button using a RadioButton element in your XML layout within a RadioGroup view group:

```
<RadioGroup
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:layout_below="@id/orderintrotext">
        <RadioButton
            android:id="@+id/sameday"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/same_day_messenger_service"
            android:onClick="onRadioButtonClicked"/>
        <RadioButton
            android:id="@+id/nextday"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/next_day_ground_delivery"
            android:onClick="onRadioButtonClicked"/>
        <RadioButton
            android:id="@+id/pickup"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/pick_up"
            android:onClick="onRadioButtonClicked"/>
    </RadioGroup>
```

Use the `android:onClick` attribute for each radio button to declare the click event handler method for the radio button, which must be defined within the activity that hosts this layout. In the above layout, clicking any radio button calls the same `onRadioButtonClicked()` method in the activity, but you could create separate methods in the activity and declare them in each radio button's `android:onClick` attribute.

The click event handler method must be `public`, return `void`, and define a `View` as its only parameter (the view that was clicked). The following shows one method, `onRadioButtonClicked()`, for all radio buttons, using `switch case` statements to check the resource `id` for the radio button element to determine which one was checked:

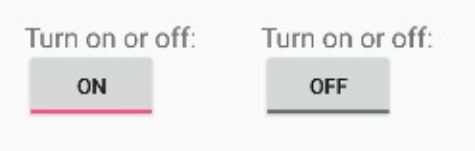
```
public void onRadioButtonClicked(View view) {
   // Check to see if a button has been clicked.
   boolean checked = ((RadioButton) view).isChecked();

   // Check which radio button was clicked.
   switch(view.getId()) {
      case R.id.sameday:
         if (checked)
            // Same day service
            break;
      case R.id.nextday:
         if (checked)
            // Next day delivery
            break;
      case R.id.pickup:
         if (checked)
            // Pick up
            break;
   }
}
```

Tip: To give users a chance to review their radio button selection before the app responds, you could implement a **Submit** or **Done** button as shown previously with checkboxes, and remove the `android:onClick` attributes from the radio buttons. Then add the `onRadioButtonClicked()` method to the `android:onClick` attribute for the **Submit** or **Done** button.

For more information about radio buttons, see "Radio Buttons" in the User Interface section of the Android Developer Documentation.

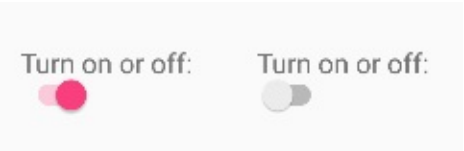
Toggle buttons and switches

A toggle input control lets the user change a setting between two states. Android provides the ToggleButton class, which shows a raised button with "OFF" and "ON".



Examples of toggles include the On/Off switches for Wi-Fi, Bluetooth, and other options in the Settings app.

Android also provides the Switch class, which is a short slider that looks like a rocker switch offering two states (on and off). Both are extensions of the CompoundButton class.



Using a toggle button

Create a toggle button by using a `ToggleButton` element in your XML layout:

```
<ToggleButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/my_toggle"
        android:text="
        android:onClick="onToggleClick"/>
```

Tip: The android:text attribute does not provide a text label for a toggle button—the toggle button always shows either "ON" or "OFF". To provide a text label next to (or above) the toggle button, use a separate `TextView`.

To respond to the toggle tap, declare an `android:onClick` callback method for the `ToggleButton`. The method must be defined in the activity hosting the layout, and it must be `public`, return `void`, and define a `View` as its only parameter (this will be the view that was clicked). Use CompoundButton.OnCheckedChangeListener() to detect the state change of the toggle. Create a `CompoundButton.OnCheckedChangeListener` object and assign it to the button by calling `setOnCheckedChangeListener()`. For example, the `onToggleClick()` method checks whether the toggle is on or off, and displays a toast message:

```
public void onToggleClick(View view) {
   ToggleButton toggle = (ToggleButton) findViewById(R.id.my_toggle);
   toggle.setOnCheckedChangeListener(new
                            CompoundButton.OnCheckedChangeListener() {
      public void onCheckedChanged(CompoundButton buttonView,
                            boolean isChecked) {
         StringBuffer onOff = new StringBuffer().append("On or off? ");
         if (isChecked) { // The toggle is enabled
            onOff.append("ON ");
         } else { // The toggle is disabled
            onOff.append("OFF ");
         }
         Toast.makeText(getApplicationContext(), onOff.toString(),
                            Toast.LENGTH_SHORT).show();
      }
   });
}
```

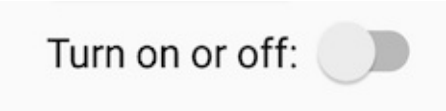
Tip: You can also programmatically change the state of a ToggleButton using the setChecked(boolean) method. Be aware, however, that the method specified by the `android:onClick()` attribute will not be executed in this case.

Using a switch

A switch is a separate instance of the Switch class, which extends the CompoundButton class just like ToggleButton. Create a toggle switch by using a `Switch` element in your XML layout:

```
<Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/my_switch"
        android:text="@string/turn_on_or_off"
        android:onClick="onSwitchClick"/>
```

The android:text attribute defines a string that appears to the left of the switch, as shown below:



To respond to the switch tap, declare an `android:onClick` callback method for the `Switch`—the code is basically the same as for a `ToggleButton`. The method must be defined in the activity hosting the layout, and it must be `public`, return `void`, and define a `View` as its only parameter (this will be the view that was clicked). Use

CompoundButton.OnCheckedChangeListener() to detect the state change of the switch. Create a `CompoundButton.OnCheckedChangeListener` object and assign it to the button by calling `setOnCheckedChangeListener()`. For example, the `onSwitchClick()` method checks whether the switch is on or off, and displays a toast message:

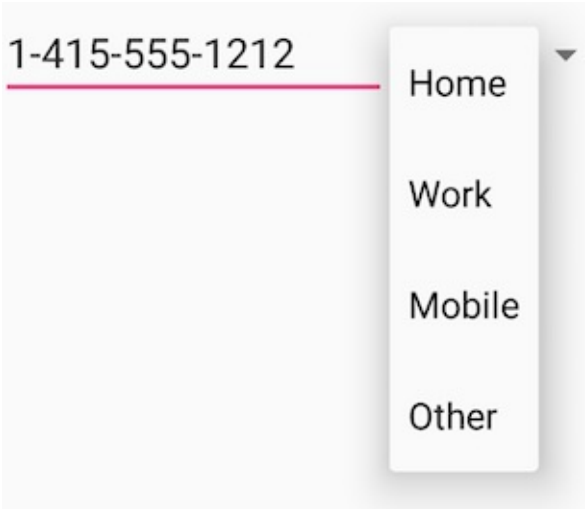
```
public void onSwitchClick(View view) {
   Switch aSwitch = (Switch) findViewById(R.id.my_switch);
   aSwitch.setOnCheckedChangeListener(new
                           CompoundButton.OnCheckedChangeListener() {
      public void onCheckedChanged(CompoundButton buttonView,
                           boolean isChecked) {
         StringBuffer onOff = new StringBuffer().append("On or off? ");
         if (isChecked) { // The switch is enabled
            onOff.append("ON ");
         } else { // The switch is disabled
            onOff.append("OFF ");
         }
         Toast.makeText(getApplicationContext(), onOff.toString(),
                           Toast.LENGTH_SHORT).show();
      }
   });
}
```

Tip: You can also programmatically change the state of a Switch using the setChecked(boolean) method. Be aware, however, that the method specified by the `android:onClick()` attribute will not be executed in this case.

For more information about toggles, see "Toggle Buttons" in the User Interface section of the Android Developer Documentation.

Spinners

A *spinner* provides a quick way to select one value from a set. Touching the spinner displays a drop-down list with all available values, from which the user can select one.



If you have a long list of choices, a spinner may extend beyond your layout, forcing the user to scroll it. A spinner scrolls automatically, with no extra code needed. However, scrolling a long list (such as a list of countries) is not recommended as it can be hard to select an item.

To create a spinner, use the Spinner class, which creates a view that displays individual spinner values as child views, and lets the user pick one. Follow these steps:

1. Create a `Spinner` element in your XML layout, and specify its values using an array and an ArrayAdapter.
2. Create the spinner and its adapter using the SpinnerAdapter class.
3. To define the selection callback for the spinner, update the Activity that uses the spinner to implement the AdapterView.OnItemSelectedListener interface.

Create the spinner UI element

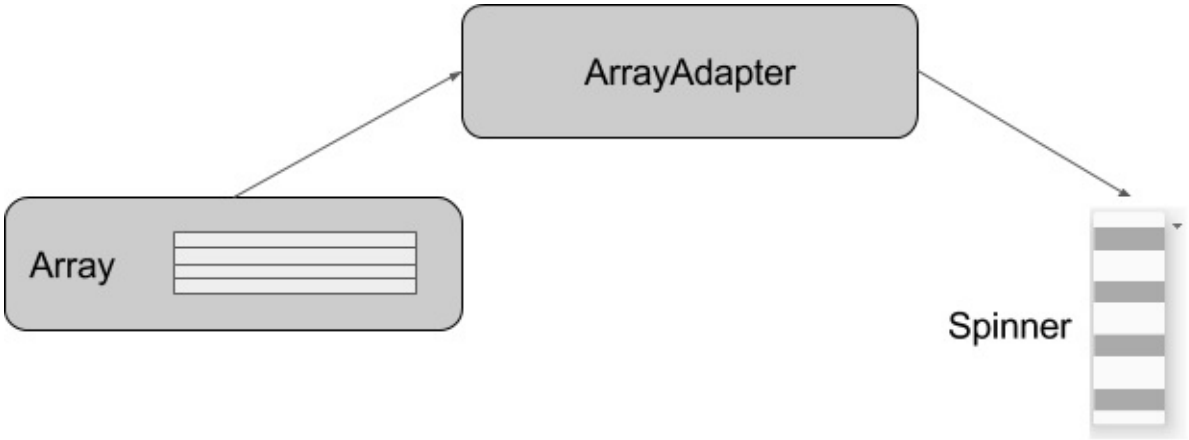
To create a spinner in your XML layout, add a Spinner element, which provides the drop-down list:

```
<Spinner
   android:id="@+id/label_spinner"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content">
</Spinner>
```

Specify the values for the spinner

You add an adapter that fills the spinner list with values. An *adapter* is like a bridge, or intermediary, between two incompatible interfaces. For example, a memory card reader acts as an adapter between the memory card and a laptop. You plug the memory card into the card reader, and plug the card reader into the laptop, so that the laptop can read the memory card.

The spinner-adapter pattern takes the data set you've specified and makes a view for each item in the data set, as shown in the figure below.



The SpinnerAdapter class, which implements the Adapter class, allows you to define two different views: one that shows the data values in the spinner itself, and one that shows the data in the drop-down list when the spinner is touched or clicked.

The values you provide for the spinner can come from any source, but must be provided through a SpinnerAdapter, such as an ArrayAdapter if the values are available in an array. The following shows a simple array called `labels_array` of predetermined values in the strings.xml file:

```
<string-array name="labels_array">
        <item>Home</item>
        <item>Work</item>
        <item>Mobile</item>
        <item>Other</item>
</string-array>
```

Tip: You can use a CursorAdapter if the values could come from a source such as a stored file or a database. You learn more about stored data in another chapter.

Create the spinner and its adapter

Create the spinner, and set its listener to the activity that implements the callback methods. The best place to do this is when the view is created in the `onCreate()` method. Follow these steps (refer to the full `onCreate()` method at the end of the steps):

1. Add the code below to the `onCreate()` method, which does the following:
2. Gets the spinner object you added to the layout using `findViewById()` to find it by its `id` (`label_spinner`).

3. Sets the onItemSelectedListener to whichever activity implements the callbacks (`this`) using the setOnItemSelectedListener() method.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   // Create the spinner.
   Spinner spinner = (Spinner) findViewById(R.id.label_spinner);
   if (spinner != null) {
      spinner.setOnItemSelectedListener(this);
   }
}
```

4. Also in the `onCreate()` method, add a statement that creates the ArrayAdapter with the string array:

```
// Create ArrayAdapter using the string array and default spinner layout.
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
            R.array.labels_array, android.R.layout.simple_spinner_item);
```

   As shown above, you use the createFromResource() method, which takes as arguments:

5. The activity that implements the callbacks for processing the results of the spinner (`this`)
6. The array (`labels_array`)
7. The layout for each spinner item (`layout.simple_spinner_item`).

   Tip: You should use the `simple_spinner_item` default layout, unless you want to define your own layout for the items in the spinner.

8. Specify the layout the adapter should use to display the list of spinner choices by calling the setDropDownViewResource() method of the ArrayAdapter class. For example, you can use `simple_spinner_dropdown_item` as your layout:

```
// Specify the layout to use when the list of choices appears.
adapter.setDropDownViewResource
                  (android.R.layout.simple_spinner_dropdown_item);
```

   Tip: You should use the `simple_spinner_dropdown_item` default layout, unless you want to define your own layout for the spinner's appearance.

9. Use setAdapter() to apply the adapter to the spinner:

```
// Apply the adapter to the spinner.
spinner.setAdapter(adapter);
```

The full code for the `onCreate()` method is shown below:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   // Create the spinner.
   Spinner spinner = (Spinner) findViewById(R.id.label_spinner);
   if (spinner != null) {
           spinner.setOnItemSelectedListener(this);
   }

   // Create ArrayAdapter using the string array and default spinner layout.
   ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
               R.array.labels_array, android.R.layout.simple_spinner_item);

   // Specify the layout to use when the list of choices appears.
   adapter.setDropDownViewResource
               (android.R.layout.simple_spinner_dropdown_item);

   // Apply the adapter to the spinner.
   if (spinner != null) {
           spinner.setAdapter(adapter);
   }
}
```

Implement the OnItemSelectedListener interface in the Activity

To define the selection callback for the spinner, update the Activity that uses the spinner to implement the AdapterView.OnItemSelectedListener interface:

```
public class MainActivity extends AppCompatActivity implements
           AdapterView.OnItemSelectedListener {
```

Android Studio automatically imports the AdapterView widget. Implement the `AdapterView.OnItemSelectedListener` interface in order to have the `onItemSelected()` and `onNothingSelected()` callback methods to use with the spinner object.

When the user chooses an item from the spinner's drop-down list, here's what happens and how you retrieve the item:

1. The `Spinner` object receives an on-item-selected event.
2. The event triggers the calling of the onItemSelected() callback method of the AdapterView.OnItemSelectedListener interface.
3. Retrieve the selected item in the spinner menu using the getItemAtPosition() method of the AdapterView class:

   ```
   public void onItemSelected(AdapterView<?> adapterView, View view, int
                  pos, long id) {
      String spinner_item = adapterView.getItemAtPosition(pos).toString();
   }
   ```

   The arguments for `onItemSelected()` are as follows:

`parent AdapterView`	The AdapterView where the selection happened
`view View`	The view within the AdapterView that was clicked
`int pos`	The position of the view in the adapter
`long id`	The row id of the item that is selected

4. Implement/override the onNothingSelected() callback method of the AdapterView.OnItemSelectedListener interface to do something if nothing is selected.

For more information about spinners, see Spinners in the "User Interface" section of the Android Developer Documentation.

Text input

Use the EditText class to get user input that consists of textual characters, including numbers and symbols. EditText extends the TextView class, to make the TextView editable.

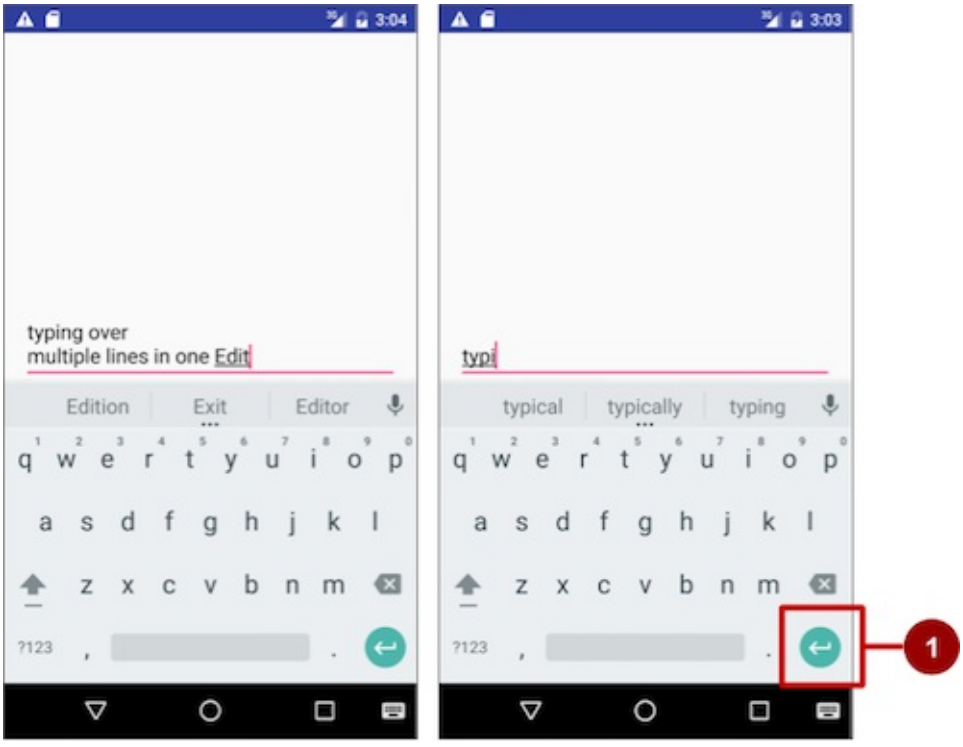
Customizing an EditText object for user input

In the Layout Manager of Android Studio, create an EditText view by adding an `EditText` to your layout with the following XML:

```
<EditText
    android:id="@+id/edit_simple"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
</EditText>
```

Enabling multiple lines of input

By default, the EditText view allows multiple lines of input as shown in the figure below, and suggests spelling corrections. Tapping the Return (also known as Enter) key on the on-screen keyboard ends a line and starts a new line in the same EditText view.



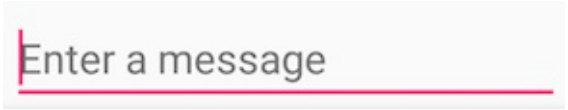
Note: In the above figure, #1 is the Return (also known as Enter) key.

Enabling Return to advance to the next view

If you add the `android:inputType` attribute to the EditText view with a value such as `"textCapCharacters"` (to change the input to all capital letters) or `"textAutoComplete"` (to enable spelling suggestions as the user types), tapping the Return key closes the on-screen keyboard and advances the focus to the next view. This behavior is useful if you want the user to fill out a form consisting of EditText fields, so that the user can advance quickly to the next EditText view.

Attributes for customizing an EditText view

Use attributes to customize the EditText view for input. For example:

- `android:maxLines="1"` : Set the text entry to show only one line.
- `android:lines="2"` : Set the text entry to show 2 lines, even if the length of the text is less.
- `android:maxLength="5"` : Set the maximum number of input characters to 5.
- `android:inputType="number"` : Restrict text entry to numbers.
- `android:digits="01"` : Restrict the digits entered to just "0" and "1".
- `android:textColorHighlight="#7cff88"` : Set the background color of selected (highlighted) text.
- `android:hint="@string/my_hint"` : Set text to appear in the field that provides a hint for the user, such as "Enter a message". 

For a list of EditText attributes, including inherited TextView attributes, see the "Summary" of the EditText class description.

Getting the user's input

Enabling the user to input text is only useful if you can *use* that text in some way in your app. To use the input, you must first get it from the EditText view in the XML layout. The steps you follow to set up the EditText view and get user input from it are:

1. Create the EditText view element in the XML layout for an activity. Be sure to identify this element with an `android:id` so that you can refer to it by its `id` :

   ```
   android:id="@+id/editText_main"
   ```

2. In the Java code for the same activity, create a method with a `View` parameter that gets the `EditText` object (in the example below, `editText`) for the `EditText` view, using the findViewById() method of the View class to find the view by its id (`editText_main`):

   ```
   EditText editText = (EditText) findViewById(R.id.editText_main);
   ```

3. Use the getText() method of the EditText class (inherited from the TextView class) to obtain the text as a character sequence (CharSequence). You can convert the character sequence into a string using the toString() method of the CharSequence class, which returns a string representing the data in the character sequence.

   ```
   String showString = editText.getText().toString();
   ```

Tip: You can use the valueOf() method of the Integer class to convert the string to an integer if the input is an integer.

Changing keyboards and input behaviors

The Android system shows an on-screen keyboard—known as a *soft* input method—when a text field in the UI receives focus. To provide the best user experience, you can specify characteristics about the type of input the app expects, such as whether it's a phone number or email address. You can also specify how the input method should behave, such as whether or not it shows spelling suggestions or provides capital letters for the beginning of a sentence. You can change the soft input method to a numeric keypad for entering only numbers, or even a phone keypad for phone numbers.

Android also provides an extensible framework for advanced programmers to develop and install their own Input Method Editors (IME) for speech input, specific types of keyboard entry, and other applications.

Declare the input method by adding the android:inputType attribute to the EditText view. For example, the following attribute sets the on-screen keyboard to be a phone keypad:

```
android:inputType="phone"
```

Use the android:inputType attribute with the following values:

- `textCapSentences` : Set the keyboard to capital letters at the beginning of a sentence.
- `textAutoCorrect` : Enable spelling suggestions as the user types.
- `textPassword` : Turn each character the user enters into a dot to conceal an entered password.
- `textEmailAddress` : For email entry, show an email keyboard with the "@" symbol conveniently located next to the space key.
- `phone` : For phone number entry, show a numeric phone keypad.

Tip: You can use the pipe `(|)` character (Java bitwise OR) to combine attribute values for the `android:inputType` attribute:

```
android:inputType="textAutoCorrect|textCapSentences"
```

For details about the `android:inputType` attribute, see Specifying the Input Method Type in the developer documentation. For a complete list of constant values for android:inputType, see the "android:inputType" section of the TextView documentation.

Changing the "action" key in the keyboard

On Android devices, the "action" key is the **Return** key. This key is normally used to enter another line of text for an EditText element that allows multiple lines. If you set an `android:inputType` attribute for the EditText view with a value such as `"textCapCharacters"` (to change the input to all capital letters) or `"textAutoComplete"` (to enable spelling suggestions as the user types), the **Return** key closes the on-screen keyboard and advances the focus to the next view.

If you want the user to enter something other than text, such as a phone number, you may want to change the "action" key to an icon for a **Send** key, and change the action to be dialing a phone number. Follow these steps:

1. Use the `android:inputType` attribute to set an input type for the keyboard:

   ```
   <EditText
      android:id="@+id/phone_number"
      android:inputType="phone"
      ... >
   </EditText>
   ```

   The `android:inputType` attribute, in the above example, sets the keyboard type to `phone` , which forces one line of input (for a phone number).

2. Use `setOnEditorActionListener()` to set the listener for the EditText view to respond to the use of the "action" key:

   ```
   EditText editText = (EditText) findViewById(R.id.phone_number);
   editText.setOnEditorActionListener(new
                              TextView.OnEditorActionListener() {
      // Add onEditorAction() method
   }
   ```

3. Use the `IME_ACTION_SEND` constant in the EditorInfo class for the `actionId` to show a **Send** key as the "action" key, and create a method to respond to the pressed **Send** key (in this case, `dialNumber` to dial the entered phone number):

   ```
   @Override
   public boolean onEditorAction(TextView textView,
                                 int actionId, KeyEvent keyEvent) {
      boolean handled = false;
      if (actionId == EditorInfo.IME_ACTION_SEND) {
         dialNumber();
         handled = true;
      }
      return handled;
   });
   ```

   Note: For help setting the listener, see "Specifying the Input ActionSpecifying Keyboard Actions" in Text Fields. For

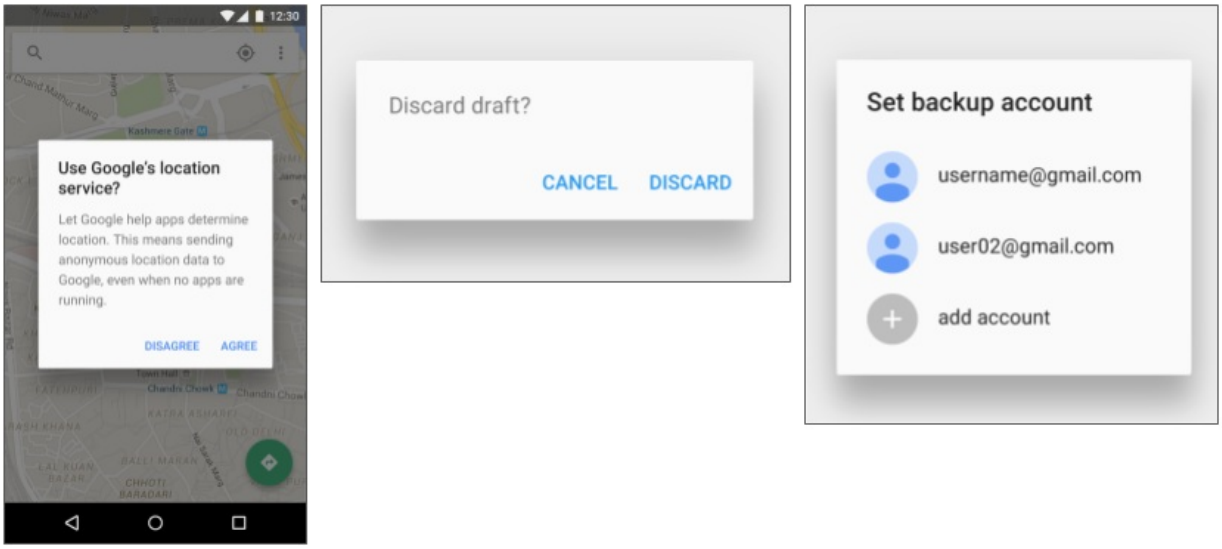
more information about the EditorInfo class, see the EditorInfo documentation.

Using dialogs and pickers

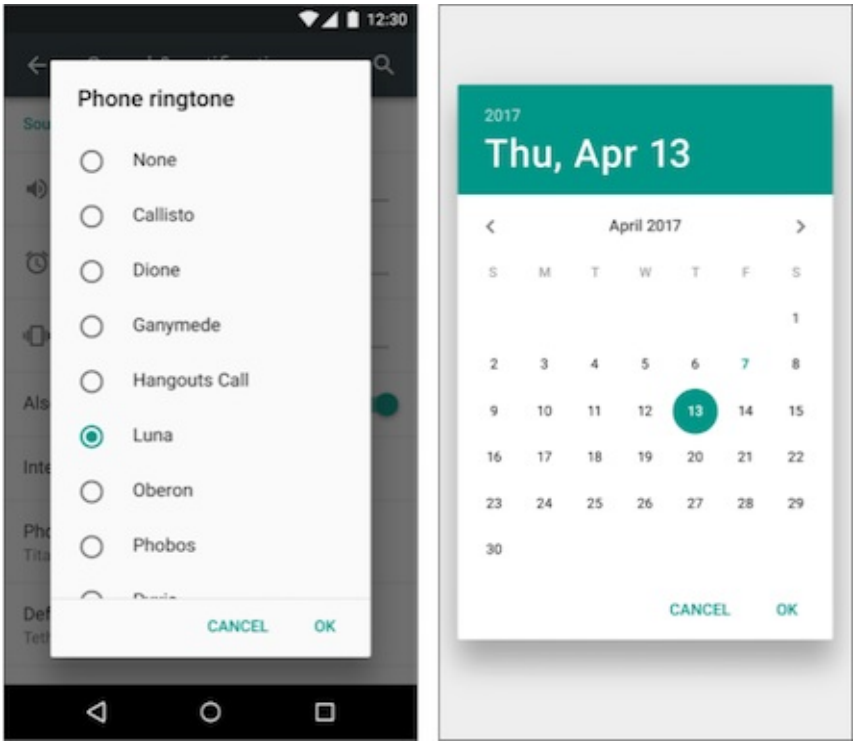
A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of activity. Dialogs inform users about a specific task and may contain critical information, require decisions, or involve multiple tasks.

For example, you would typically use a dialog to show an alert that requires users to tap a button make a decision, such as **OK** or **Cancel**. In the figure below, the left side shows an alert with **Disagree** and **Agree** buttons, and the center shows an alert with **Cancel** and **Discard** buttons.

You can also use a dialog to provide choices in the style of radio buttons, as shown on the right side of the figure below.



The base class for all dialog components is a Dialog. There are several useful Dialog subclasses for alerting the user on a condition, showing status or progress, displaying information on a secondary device, or selecting or confirming a choice, as shown on the left side of the figure below. The Android SDK also provides ready-to-use dialog subclasses such as *pickers* for picking a time or a date, as shown on the right side of the figure below. Pickers allow users to enter information in a predetermined, consistent format that reduces the chance for input error.



Dialogs always retain focus until dismissed or a required action has been taken.

Tip: Best practices recommend using dialogs sparingly as they interrupt the user's work flow. Read the Dialogs design guide for additional best design practices, and Dialogs in the Android developer documentation for code examples.

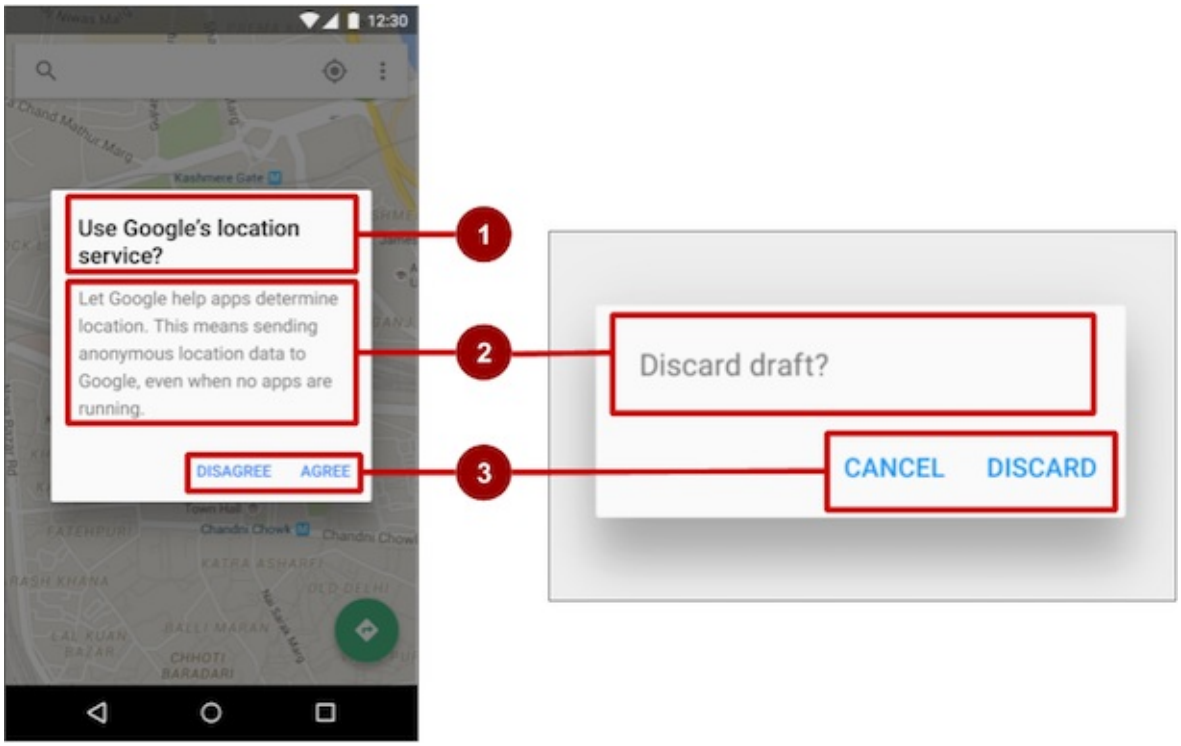
The Dialog class is the base class for dialogs, but you should avoid instantiating Dialog directly unless you are creating a custom dialog. For standard Android dialogs, use one of the following subclasses:

- AlertDialog: A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- DatePickerDialog or TimePickerDialog: A dialog with a pre-defined UI that lets the user select a date or time.

Showing an alert dialog

Alerts are urgent interruptions, requiring acknowledgement or action, that inform the user about a situation as it occurs, or an action *before* it occurs (as in discarding a draft). You can provide buttons in an alert to make a decision. For example, an alert dialog might require the user to click **Continue** after reading it, or give the user a choice to agree with an action by clicking a positive button (such as **OK** or **Accept**), or to disagree by clicking a negative button (such as **Disagree** or **Cancel**).

Use the AlertDialog subclass of the Dialog class to show a standard dialog for an alert. The AlertDialog class allows you to build a variety of dialog designs. An alert dialog can have the following regions (refer to the diagram below):



1. Title: A title is optional. Most alerts don't need titles. If you can summarize a decision in a sentence or two by either asking a question (such as, "Discard draft?") or making a statement related to the action buttons (such as, "Click OK to continue"), don't bother with a title. Use a title if the situation is high-risk, such as the potential loss of connectivity or data, and the content area is occupied by a detailed message, a list, or custom layout.
2. Content area: The content area can display a message, a list, or other custom layout.
3. Action buttons: You should use no more than three action buttons in a dialog, and most have only two.

Building the AlertDialog

The AlertDialog.Builder class uses the *builder* design pattern, which makes it easy to create an object from a class that has a lot of required and optional attributes and would therefore require a lot of parameters to build. Without this pattern, you would have to create constructors for combinations of required and optional attributes; with this pattern, the code is easier to read and maintain. For more information about the builder design pattern, see Builder pattern.

Use `AlertDialog.Builder` to build a standard alert dialog and set attributes on the dialog. Use setTitle() to set its title, setMessage() to set its message, and setPositiveButton() and setNegativeButton() to set its buttons.

Note: If `AlertDialog.Builder` is not recognized as you enter it, you may need to add the following import statements to MainActivity.java:

```
import android.content.DialogInterface;
import android.support.v7.app.AlertDialog;
```

The following creates the dialog object (`myAlertBuilder`) and sets the title (the string resource called `alert_title`) and message (the string resource called `alert_message`):

```
AlertDialog.Builder myAlertBuilder = new
                AlertDialog.Builder(MainActivity.this);
myAlertBuilder.setTitle(R.string.alert_title);
myAlertBuilder.setMessage(R.string.alert_message);
```

Setting the button actions for the alert dialog

Use the setPositiveButton() and setNegativeButton() methods of the AlertDialog.Builder class to set the button actions for the alert dialog. These methods require a title for the button (supplied by a string resource) and the DialogInterface.OnClickListener class that defines the action to take when the user presses the button:

```
myAlertBuilder.setPositiveButton("OK", new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) {
           // User clicked OK button.
      }
});
myAlertBuilder.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) {
         // User clicked the CANCEL button.
      }
   });
```

You can add only one of each button type to an AlertDialog. For example, you can't have more than one "positive" button.

Tip: You can also set a "neutral" button with setNeutralButton(). The neutral button appears between the positive and negative buttons. Use a neutral button, such as "Remind me later", if you want the user to be able to dismiss the dialog and decide later.

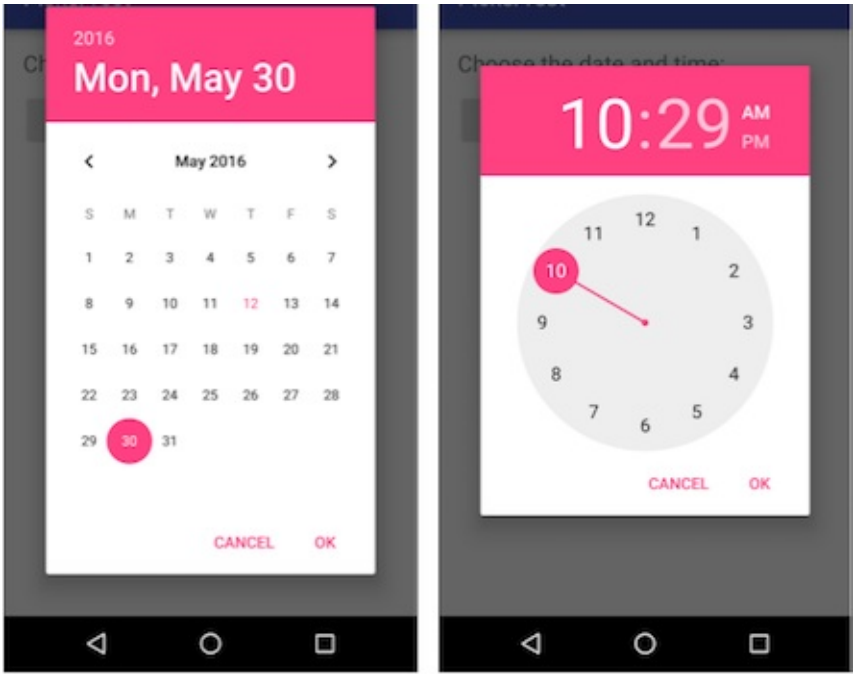
Displaying the dialog

To display the dialog, call its `show()` method:

```
alertDialog.show();
```

Date and time pickers

Android provides ready-to-use dialogs, called *pickers*, for picking a time or a date. Use them to ensure that your users pick a valid time or date that is formatted correctly and adjusted to the user's locale. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year).



When showing a picker, you should use an instance of DialogFragment, a subclass of Fragment, which displays a dialog window floating on top of its activity's window. A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own individual lifecycle. A fragment receives its own input events, and you can add or remove it while the main activity is running. You might combine multiple fragments in a single activity to build a multiple-pane user interface, or reuse a fragment in multiple activities. To learn about fragments, see Fragments in the API Guide.

One benefit of using fragments for the pickers is that you can isolate the code sections for managing the date and the time after the user selects them from the pickers. You can also use DialogFragment to manage the dialog lifecycle.

Tip: Another benefit of using fragments for the pickers is that you can implement different layout configurations, such as a basic dialog on handset-sized displays or an embedded part of a layout on large displays.

Adding a fragment

To add a fragment for the date picker, create a blank fragment (**DatePickerFragment**) without a layout XML, and without factory methods or interface callbacks:

1. Expand **app > java > com.example.android.DateTimePickers** and select **MainActivity**.
2. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **DatePickerFragment**. Uncheck all three checkbox options so that you do *not* create a layout XML, do *not* include fragment factory methods, and do *not* include interface callbacks. You don't need to create a layout for a standard picker. Click **Finish** to create the fragment.

Extending DialogFragment for the picker

The next step is to create a standard picker with a listener. Follow these steps:

1. Edit the `DatePickerFragment` class definition to extend `DialogFragment,` and implement `DatePickerDialog.OnDateSetListener` to create a standard date picker with a listener:

   ```
   public class DatePickerFragment extends DialogFragment
                       implements DatePickerDialog.OnDateSetListener {
   ...
   }
   ```

   Android Studio automatically adds the following in the import block at the top:

```
import android.app.DatePickerDialog.OnDateSetListener;
import android.support.v4.app.DialogFragment;
```

2. Android Studio also shows a red light bulb icon in the left margin, prompting you to implement methods. Click the icon and, with **onDateSet** already selected and the "Insert @Override" option checked, click **OK** to create the empty onDateSet() method.

   Android Studio then automatically adds the following in the import block at the top:

   ```
   import android.widget.DatePicker;
   ```

3. Replace `onCreateView()` with `onCreateDialog()`:

   ```
   @Override
   public Dialog onCreateDialog(Bundle savedInstanceState) {
   }
   ```

   When you extend `DialogFragment`, you should override the onCreateDialog() callback method, rather than `onCreateView`. You use your version of the callback method to initialize the `year`, `month`, and `day` for the date picker.

Setting the defaults and returning the picker

To set the default date in the picker and return it as an object you can use, follow these steps:

1. Add the following code to the `onCreateDialog()` method to set the default date for the picker:

   ```
   // Use the current date as the default date in the picker.
   final Calendar c = Calendar.getInstance();
   int year = c.get(Calendar.YEAR);
   int month = c.get(Calendar.MONTH);
   int day = c.get(Calendar.DAY_OF_MONTH);
   ```

   As you enter `Calendar`, you are given a choice of which Calendar library to import. Choose this one:

   ```
   import java.util.Calendar;
   ```

   The Calendar class sets the default date as the current date—it converts between a specific instant in time and a set of calendar fields such as `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR`, and so on. Calendar is locale-sensitive, and its class method `getInstance()` returns a Calendar object whose calendar fields have been initialized with the current date and time.

2. Add the following statement to the end of the method to create a new instance of the date picker and return it:

   ```
   return new DatePickerDialog(getActivity(), this, year, month, day);
   ```

Showing the picker

In the Main Activity, you need to create a method to show the date picker. Follow these steps:

1. Create a method to instantiate the date picker dialog fragment:

   ```
   public void showDatePickerDialog(View v) {
       DialogFragment newFragment = new DatePickerFragment();
       newFragment.show(getSupportFragmentManager(), "datePicker");
   }
   ```

2. You can then use `showDatePickerDialog()` with the `android:onClick` attribute for a button or other input control:

```
<Button
        android:id="@+id/button_date"
        ...
        android:onClick="showDatePickerDialog"/>
```

Processing the user's picker choice

The `onDateSet()` method is automatically called when the user makes a selection in the date picker, so you can use this method to do something with the chosen date. Follow these steps:

1. To make the code more readable, change the `onDateSet()` method's parameters from `int i`, `int i1`, and `int i2` to `int year`, `int month`, and `int day`:

   ```
   public void onDateSet(DatePicker view, int year, int month, int day) {
   ```

2. Open **MainActivity** and add the `processDatePickerResult()` method signature that takes the `year`, `month`, and `day` as arguments:

   ```
   public void processDatePickerResult(int year, int month, int day) {
   }
   ```

3. Add the following code to the `processDatePickerResult()` method to convert the `month`, `day`, and `year` to separate strings:

   ```
   String month_string = Integer.toString(month+1);
   String day_string = Integer.toString(day);
   String year_string = Integer.toString(year);
   ```

   Note: The `month` integer returned by the date picker starts counting at 0 for January, so you need to add 1 to it to start show months starting at 1.

4. Add the following after the above code to concatenate the three strings and include slash marks for the U.S. date format:

   ```
   String dateMessage = (month_string + "/" +
                              day_string + "/" + year_string);
   ```

5. Add the following after the above statement to display a Toast message:

   ```
   Toast.makeText(this, "Date: " + dateMessage,
                              Toast.LENGTH_SHORT).show();
   ```

6. Extract the hard-coded string `"Date: "` into a string resource named `date`. This automatically replaces the hard-coded string with `getString(R.string.date)`. The code for the `processDatePickerResult()` method should now look like this:

   ```
   public void processDatePickerResult(int year, int month, int day) {
      String month_string = Integer.toString(month + 1);
      String day_string = Integer.toString(day);
      String year_string = Integer.toString(year);
      // Assign the concatenated strings to dateMessage.
      String dateMessage = (month_string + "/" +
                                  day_string + "/" + year_string);
      Toast.makeText(this, getString(R.string.date) + dateMessage,
                                  Toast.LENGTH_SHORT).show();
   }
   ```

7. Open **DatePickerFragment**, and add the following to the `onDateSet()` method to invoke the `processDatePickerResult()` method in `MainActivity` and pass it the `year`, `month`, and `day`:

```
public void onDateSet(DatePicker view, int year, int month, int day) {
   // Set the activity to the Main Activity.
   MainActivity activity = (MainActivity) getActivity();
   // Invoke Main Activity's processDatePickerResult() method.
   activity.processDatePickerResult(year, month, day);
}
```

You use `getActivity()` which, when used in a fragment, returns the activity the fragment is currently associated with. You need this because you can't call a method in `MainActivity` without the context of `MainActivity` (you would have to use an `intent` instead, as you learned in a previous lesson). The activity inherits the context, so you can use it as the context for calling the method (as in `activity.processDatePickerResult`) .

Using the same procedures for the time picker

Follow the same procedures outlined above for a date picker:

1. Add a blank fragment called **TimePickerFragment** that extends `DialogFragment` and implements `TimePickerDialog.OnTimeSetListener`:

   ```
   public class TimePickerFragment extends DialogFragment
                   implements TimePickerDialog.OnTimeSetListener {
   ```

2. Add with `@Override` a blank `onTimeSet()` method:

   Android Studio also shows a red light bulb icon in the left margin, prompting you to implement methods. Click the icon and, with **onTimeSet** already selected and the "Insert @Override" option checked, click **OK** to create the empty onTimeSet() method. Android Studio then automatically adds the following in the import block at the top:

   ```
   import android.widget.TimePicker;
   ```

3. Use `onCreateDialog()` to initialize the time and return the dialog:

   ```
   public Dialog onCreateDialog(Bundle savedInstanceState) {
           // Use the current time as the default values for the picker.
           final Calendar c = Calendar.getInstance();
           int hour = c.get(Calendar.HOUR_OF_DAY);
           int minute = c.get(Calendar.MINUTE);

           // Create a new instance of TimePickerDialog and return it.
           return new TimePickerDialog(getActivity(), this, hour, minute,
                   DateFormat.is24HourFormat(getActivity()));
       }
   ```

4. Show the picker: Open **MainActivity** and create a method to instantiate the date picker dialog fragment:

   ```
   public void showDatePickerDialog(View v) {
           DialogFragment newFragment = new DatePickerFragment();
           newFragment.show(getSupportFragmentManager(), "datePicker");
   }
   ```

   Use `showDatePickerDialog()` with the `android:onClick` attribute for a button or other input control:

   ```
   <Button
           android:id="@+id/button_date"
           ...
           android:onClick="showDatePickerDialog"/>
   ```

5. Create the `processTimePickerResult()` method in **MainActivity** to process the result of choosing from the time picker:

```
public void processTimePickerResult(int hourOfDay, int minute) {
   // Convert time elements into strings.
   String hour_string = Integer.toString(hourOfDay);
   String minute_string = Integer.toString(minute);
   // Assign the concatenated strings to timeMessage.
   String timeMessage = (hour_string + ":" + minute_string);
   Toast.makeText(this, getString(R.string.time) + timeMessage,
                                  Toast.LENGTH_SHORT).show();
}
```

6. Use `onTimeSet()` to get the time and pass it to the `processTimePickerResult()` method in MainActivity:

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
   // Set the activity to the Main Activity.
   MainActivity activity = (MainActivity) getActivity();
   // Invoke Main Activity's processTimePickerResult() method.
   activity.processTimePickerResult(hourOfDay, minute);
}
```

You can read all about setting up pickers in Pickers.

Recognizing gestures

A *touch gesture* occurs when a user places one or more fingers on the touch screen, and your app interprets that pattern of touches as a particular gesture, such as a long touch, double-tap, fling, or scroll.

Android provides a variety of classes and methods to help you create and detect gestures. Although your app should not depend on touch gestures for basic behaviors (since the gestures may not be available to all users in all contexts), adding touch-based interaction to your app can greatly increase its usefulness and appeal.

To provide users with a consistent, intuitive experience, your app should follow the accepted Android conventions for touch gestures. The Gestures design guide shows you how to design common gestures in Android apps. For more code samples and details, see Using Touch Gestures in the Android developer documentation.

Detecting common gestures

If your app uses common gestures such as double tap, long press, fling, and so on, you can take advantage of the GestureDetector class for detecting common gestures. Use GestureDetectorCompat, which is provided as a compatibility implementation of the framework's GestureDetector class which guarantees the newer focal point scrolling behavior from Jellybean MR1 on all platform versions. This class should be used only with motion events reported for touch devices—don't use it for trackball or other hardware events.

GestureDetectorCompat lets you detect common gestures without processing the individual touch events yourself. It detects various gestures and events using MotionEvent objects, which report movements by a finger (or mouse, pen, or trackball).

The following snippets show how you would use `GestureDetectorCompat` and the `GestureDetector.SimpleOnGestureListener` class.

1. To use GestureDetectorCompat, create an instance (`mDetector` in the snippet below) of the `GestureDetectorCompat` class, using the `onCreate()` method in the activity (such as **MainActivity**):

```
public class MainActivity extends Activity {
    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new
            MyGestureListener());
    }
    ...
}
```

When you instantiate a `GestureDetectorCompat` object, one of the parameters it takes is a class you must create — `MyGestureListener` in the above snippet—that does one of the following:

- implements the GestureDetector.OnGestureListener interface, to detect all standard gestures, or
- extends the GestureDetector.SimpleOnGestureListener class, which you can use to process only a few gestures by overriding the methods you need. Some of the methods it provides include onDown(), onLongPress(), onFling(), onScroll(), and onSingleTapUp().

1. Create the class `MyGestureListener` as a separate activity (**MyGestureListener**) to extend `GestureDetector.SimpleOnGestureListener`, and override the onFling() and onDown() methods to show log statements about the event:

```
class MyGestureListener
                    extends GestureDetector.SimpleOnGestureListener {
   private static final String DEBUG_TAG = "Gestures";

   @Override
   public boolean onDown(MotionEvent event) {
      Log.d(DEBUG_TAG,"onDown: " + event.toString());
      return true;
   }

   @Override
   public boolean onFling(MotionEvent event1, MotionEvent event2,
                  float velocityX, float velocityY) {
       Log.d(DEBUG_TAG, "onFling: " +
            event1.toString()+event2.toString());
       return true;
   }
}
```

2. To intercept touch events, override the onTouchEvent() callback of the GestureDetectorCompat class in **MainActivity**:

```
@Override
public boolean onTouchEvent(MotionEvent event){
  this.mDetector.onTouchEvent(event);
  return super.onTouchEvent(event);
}
```

Detecting all gestures

To detect all types of gestures, you need to perform two essential steps:

1. Gather data about touch events.
2. Interpret the data to see if it meets the criteria for any of the gestures your app supports.

The gesture starts when the user first touches the screen, continues as the system tracks the position of the user's finger(s), and ends by capturing the final event of the user's fingers leaving the screen. Throughout this interaction, an object of the MotionEvent class is delivered to onTouchEvent(), providing the details of every interaction. Your app can use the data provided by the MotionEvent to determine if a gesture it cares about happened.

For example, when the user first touches the screen, the onTouchEvent() method is triggered on the view that was touched, and a MotionEvent object reports movement by a finger (or mouse, pen, or trackball) in terms of:

- *An action code:* Specifies the state change that occurred, such as a finger tapping down or lifting up.
- *A set of axis values:* Describes the position in X and Y coordinates of the touch and information about the pressure, size and orientation of the contact area.

The individual fingers or other objects that generate movement traces are referred to as *pointers*. Some devices can report multiple movement traces at the same time. Multi-touch screens show one movement trace for each finger. Motion events contain information about all of the pointers that are currently active even if some of them have not moved since the last event was delivered. Based on the interpretation of the MotionEvent object, the onTouchEvent() method triggers the appropriate callback on the GestureDetector.OnGestureListener interface.

Each MotionEvent pointer has a unique id that is assigned when it first goes down (indicated by `ACTION_DOWN` or `ACTION_POINTER_DOWN)`. A pointer id remains valid until the pointer eventually goes up (indicated by `ACTION_UP` or `ACTION_POINTER_UP`) or when the gesture is canceled (indicated by `ACTION_CANCEL`). The MotionEvent class provides methods to query the position and other properties of pointers, such as getX(int), getY(int), getAxisValue(int), getPointerId(int), and getToolType(int).

The interpretation of the contents of a MotionEvent varies significantly depending on the source class of the device. On touch screens, the pointer coordinates specify absolute positions such as view X/Y coordinates. Each complete gesture is represented by a sequence of motion events with actions that describe pointer state transitions and movements.

A gesture starts with a motion event with `ACTION_DOWN` that provides the location of the first pointer down. As each additional pointer goes down or up, the framework generates a motion event with `ACTION_POINTER_DOWN` or `ACTION_POINTER_UP` accordingly. Pointer movements are described by motion events with `ACTION_MOVE` . A gesture ends when the final pointer goes up as represented by a motion event with `ACTION_UP` , or when the gesture is canceled with `ACTION_CANCEL` .

To intercept touch events in an activity or view, override the `onTouchEvent()` callback as shown in the snippet below. You can use the getActionMasked() method of the MotionEventCompat class to extract the action the user performed from the event parameter. (MotionEventCompat is a helper for accessing features in a MotionEvent, which was introduced after API level 4 in a backwards compatible fashion.) This gives you the raw data you need to determine if a gesture you care about occurred:

```
public class MainActivity extends Activity {
...
// This example shows an Activity, but you would use the same approach if
// you were subclassing a View.
   @Override
   public boolean onTouchEvent(MotionEvent event){

       int action = MotionEventCompat.getActionMasked(event);

       switch(action) {
          case (MotionEvent.ACTION_DOWN) :
             Log.d(DEBUG_TAG,"Action was DOWN");
             return true;
          case (MotionEvent.ACTION_MOVE) :
             Log.d(DEBUG_TAG,"Action was MOVE");
             return true;
          case (MotionEvent.ACTION_UP) :
             Log.d(DEBUG_TAG,"Action was UP");
             return true;
          case (MotionEvent.ACTION_CANCEL) :
             Log.d(DEBUG_TAG,"Action was CANCEL");
             return true;
          case (MotionEvent.ACTION_OUTSIDE) :
             Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
             return true;
          default :
             return super.onTouchEvent(event);
    }
}
```

You can then do your own processing on these events to determine if a gesture occurred.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Using Keyboards, Input Controls, Alerts, and Pickers

Learn more

- Android API Guide, "Develop" section:
  - Specifying the Input Method Type
  - Handling Keyboard Input
  - Text Fields
  - EditorInfo
  - Input Controls
  - Buttons
  - Styles and Themes
  - Spinners
  - Dialogs
  - Fragments
  - Input Events
  - Pickers
  - DateFormat
  - Using Touch Gestures
- Material Design Spec:
  - Components - Buttons
  - Dialogs design guide

    - Gestures design guide
- Developer Blog:
    - "Holo Everywhere"
    - Implementing Material Design in Your Android app

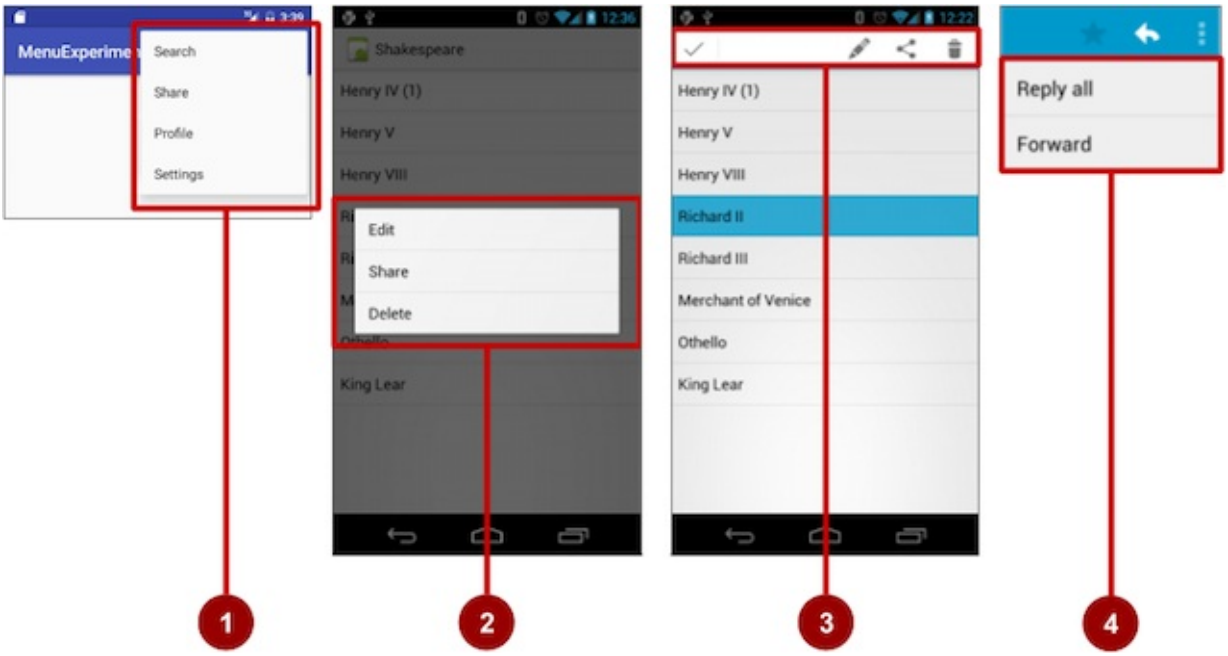
4.2: Menus

Contents:

- Types of menus
- The app bar and options menu
- Contextual menu
- Popup menu
- Related practical
- Learn more

Types of menus

A menu is a set of options the user can select from to perform a function, such as searching for information, saving information, editing information, or navigating to a screen. Android offers the following types of menus, which are useful for different situations (refer to the figure below):



1. *Options menu:* Appears in the app bar and provides the primary options that affect using the app itself. Examples of menu options: **Search** to perform a search, **Bookmark** to save a link to a screen, and **Settings** to navigate to the Settings screen.
2. *Context menu:* Appears as a floating list of choices when the user performs a long tap on an element on the screen. Examples of menu options: **Edit** to edit the element, **Delete** to delete it, and **Share** to share it over social media.
3. *Contextual action bar:* Appears at the top of the screen overlaying the app bar, with action items that affect the selected element(s). Examples of menu options: **Edit**, **Share**, and **Delete** for one or more selected elements.
4. *Popup menu:* Appears anchored to a view such as an ImageButton, and provides an overflow of actions or the second part of a two-part command. Example of a popup menu: the Gmail app anchors a popup menu to the app bar for the message view with **Reply**, **Reply All**, and **Forward**.

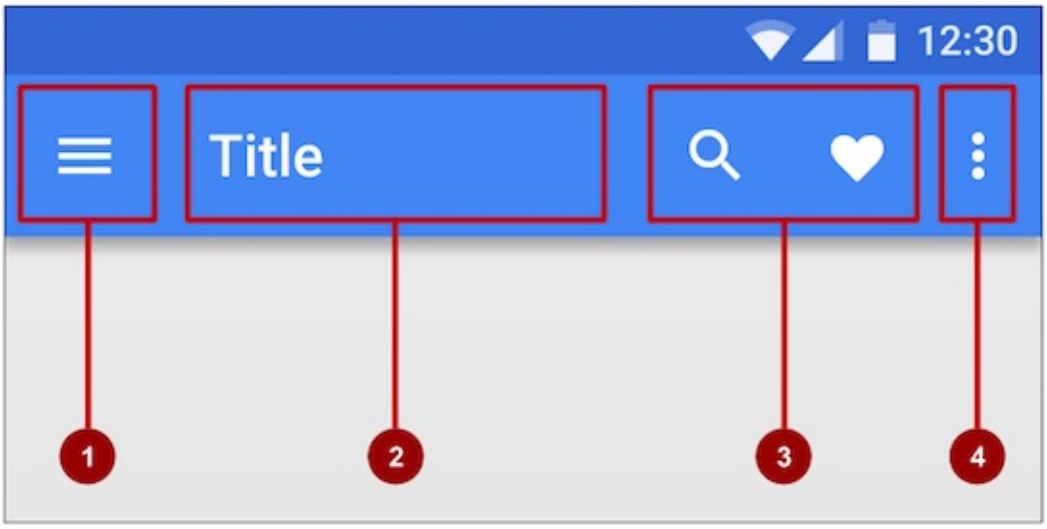
The app bar and options menu

The *app bar* (also called the *action bar*) is a dedicated space at the top of each activity screen. When you create an activity from a template (such as Empty Template), an app bar is automatically included for the activity in a CoordinatorLayout root view group at the top of the view hierarchy.

The app bar by default shows the app title, or the name defined in AndroidManifest.xml by the `android:label` attribute for the activity. It may also include the **Up** button for navigating up to the parent activity, which is described in the next chapter.

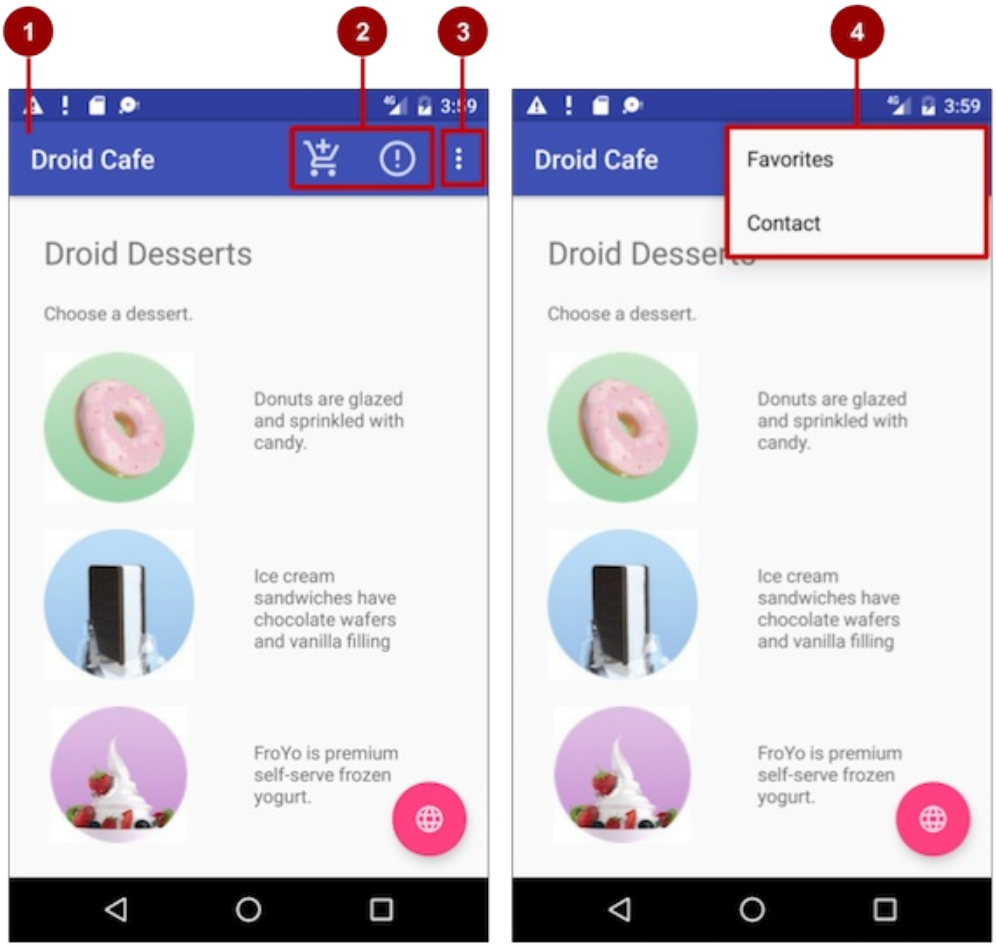
The *options menu* in the app bar provides navigation to other activities in the app, or the primary options that affect using the app itself — but not ones that perform an action on an element on the screen. For example, your options menu might provide the user choices for navigating to other activities, such as placing an order, or for actions that have a global impact on the app, such as changing settings or profile information.

The options menu appears in the right corner of the app bar. The app bar is split into four different functional areas that apply to most apps:



1. *Navigation button or Up button:* Use a navigation button in this space to open a navigation drawer, or use an Up button for navigating up through your app's screen hierarchy to the parent activity. Both are described in the next chapter.
2. *Title:* The title in the app bar is the app title, or the name defined in AndroidManifest.xml by the `android:label` attribute for the activity.
3. *Action icons for the options menu:* Each action icon appears in the app bar and represents one of the options menu's most frequently used items. Less frequently used options menu items appear in the overflow options menu.
4. *Overflow options menu:* The overflow icon opens a popup with option menu items that are not shown as icons in the app bar.

Frequently-used options menu items should appear as icons in the app bar. The overflow options menu shows the rest of the menu:



In the above figure:

1. **App bar**. The app bar includes the app title, the options menu, and the overflow button.
2. **Options menu action icons**. The first two options menu items appear as icons in the app bar.
3. **Overflow button**. The overflow button (three vertical dots) opens a menu that shows more options menu items.
4. **Options overflow menu**. After clicking the overflow button, more options menu items appear in the overflow menu.

Adding the app bar

Each activity that uses the default theme also has an ActionBar as its app bar. Some themes also set up an ActionBar as an app bar by default. When you start an app from a template such as Empty Activity, an ActionBar appears as the app bar.

However, as features were added to the native ActionBar over various Android releases, the native ActionBar behaves differently depending on the version of Android running on the device. For this reason, if you are adding an options menu, you should use the v7 appcompat support library's Toolbar as an app bar. Using the Toolbar makes it easy to set up an app bar that works on the widest range of devices, and also gives you room to customize your app bar later on as your app develops. Toolbar includes the most recent features, and works for any device that can use the support library.

In order to use Toolbar as the app bar (rather than the default ActionBar) for an activity, do one of the following:

- Start your project with the Basic Activity template, which implements the Toolbar for the activity as well as a rudimentary options menu (with one item, **Settings**). You can skip this section.
- Do it yourself, as shown in this section:
    1. Add the support libraries: appcompat and design.
    2. Use a NoActionBar theme and styles for the app bar and background.
    3. Add an AppBarLayout and a Toolbar to the layout.
    4. Add code to the activity to set up the app bar.

Adding the support libraries

If you start an app project using the Basic Activity template, the template adds the following support libraries for you, so you can skip this step.

If you are *not* using the Basic Activity template, add the appcompat support library (current version is v7) for the Toolbar class, and the design library for the NoActionBar themes, to your project:

1. Choose **Tools > Android > SDK Manager** to check that the Android Support Repository is installed; if it is not, install it.
2. Open the build.gradle file for your app, and add the support library feature project identifiers to the `dependencies` section. For example, to include `support:appcompat` and `support:design`, add:

```
compile 'com.android.support:appcompat-v7:23.4.0'
compile 'com.android.support:design:23.4.0'
```

Note: Update the version numbers for dependencies if necessary. If the version number you specified is lower than the currently available library version number, Android Studio will warn you ("a newer version of com.android.support:design is available"). Update the version number to the one Android Studio tells you to use.

Using themes to design the app bar

If you start an app project using the Basic Activity template, the template adds the theme to replace the ActionBar with a `Toolbar`, so you can skip this step.

If you are *not* using the Basic Activity template, you can use the `Toolbar` class for the app bar by turning off the default ActionBar using a NoActionBar theme for the activity. Themes in Android are similar to styles, except that they are applied to an entire app or activity rather than to a specific view.

When you create a new project in Android Studio, an app theme is automatically generated for you. For example, if you start an app project with the Empty Activity or Basic Activity template, the `AppTheme` theme is provided in **styles.xml** in the **res > values** directory.

Tip: You learn more about themes in the chapter on drawables, styles and themes.

You can modify the theme to provide a style for the app bar and app background so that the app bar is visible and stands out against the background. Follow these steps:

1. Open the **styles.xml** file. You should already have the following in the file:

```
<resources>
   <!-- Base application theme. -->
   <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
      <!-- Customize your theme here. -->
      <item name="colorPrimary">@color/colorPrimary</item>
      <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
      <item name="colorAccent">@color/colorAccent</item>
   </style>
   ...
</resources>
```

`AppTheme` "inherits"—takes on all the styles—from a parent them called `Theme.AppCompat.Light.DarkActionBar`, which is a standard theme supplied with Android. However, you can override an inherited style with another style by adding the other style to **styles.xml**.

2. Add the `AppTheme.NoActionBar`, `AppTheme.AppBarOverlay`, and `AppTheme.PopupOverlay` styles under the `AppTheme` style, as shown below. These styles will override the style attributes with the same names in `AppTheme`, affecting the appearance of the app bar and the app's background:

```
<resources>
   <!-- Base application theme. -->
   <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
      ...
   </style>

   <style name="AppTheme.NoActionBar">
      <item name="windowActionBar">false</item>
      <item name="windowNoTitle">true</item>
   </style>

   <style name="AppTheme.AppBarOverlay"
            parent="ThemeOverlay.AppCompat.Dark.ActionBar" />

   <style name="AppTheme.PopupOverlay"
            parent="ThemeOverlay.AppCompat.Light" />
   ...
</resources>
```

3. In the AndroidManifest.xml file, add the NoActionBar theme in appcompat to the `<application>` element. Using this theme prevents the app from using the native ActionBar class to provide the app bar:

```
<activity
   ...
   android:theme="@style/AppTheme.NoActionBar">
</activity>
```

Adding AppBarLayout and a Toolbar to the layout

If you start an app project using the Basic Activity template, the template adds the `AppBarLayout` and `Toolbar` for you, so you can skip this step.

If you are *not* using the Basic Activity template, you can include the Toolbar in an activity's layout by adding an `AppBarLayout` and a `Toolbar` element. AppBarLayout is a vertical LinearLayout which implements many of the features of the material designs app bar concept, such as scrolling gestures. Keep in mind the following:

- `AppBarLayout` must be a direct child within a `CoordinatorLayout` root view group, and `Toolbar` must be a direct child within `AppBarLayout`, as shown below:

```
<android.support.design.widget.CoordinatorLayout
   ... >
   <android.support.design.widget.AppBarLayout
      ...>
      <android.support.v7.widget.Toolbar
            ...
            />
   </android.support.design.widget.AppBarLayout>
   ...
</android.support.design.widget.CoordinatorLayout>
```

- Position the `Toolbar` at the top of the activity's layout, since you are using it as an app bar.
- `AppBarLayout` also requires a separate content layout sibling for the content that scrolls underneath the app bar. You can add this sibling as a view group (such as RelativeLayout or LinearLayout) as follows:
  - In the same layout file for the activity (as in **activity_main.xml**)
  - In a separate layout file, such as **content_main.xml**, which you can then add to the activity's layout file with an include statement:

    ```
    <include layout="@layout/content_main" />
    ```

- You need to set the content sibling's scrolling behavior, as shown below with the RelativeLayout group, to be an instance of `AppBarLayout.ScrollingViewBehavior`:

```
<RelativeLayout
   ...
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   app:layout_behavior="@string/appbar_scrolling_view_behavior"
   ... >

</RelativeLayout>
```

The layout *behavior* for the `RelativeLayout` is set to the string resource `@string/appbar_scrolling_view_behavior`, which controls the scrolling behavior of the screen in relation to the app bar at the top. This string resource represents the following string, which is defined in the values.xml file that should not be edited:

```
android.support.design.widget.AppBarLayout$ScrollingViewBehavior
```

This behavior is defined by the AppBarLayout.ScrollingViewBehavior class. This behavior should be used by Views which can scroll vertically—it supports nested scrolling to automatically scroll any AppBarLayout siblings.

Adding code to set up the app bar

If you start an app project using the Basic Activity template, the template adds the code needed to set up the app bar, so you can skip this step.

If you are *not* using the Basic Activity template, you can follow these steps to set up the app bar in the activity:

1. Make sure that any activity that you want to show an app bar extends AppCompatActivity:

   ```
   public class MyActivity extends AppCompatActivity {
   ...
   }
   ```

2. In the activity's `onCreate()` method, call the activity's setSupportActionBar() method, and pass the activity's toolbar (assuming the Toolbar element's id is `toolbar`). The `setSupportActionBar()` method sets the toolbar as the app bar for the activity:

   ```
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
      setSupportActionBar(toolbar);
   }
   ```

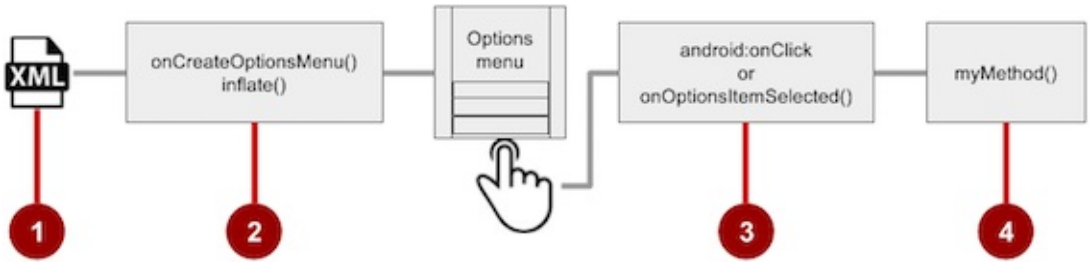
The activity now shows the app bar. By default, the app bar contains just the name of the app.

Adding the options menu

Android provides a standard XML format to define options menu items. Instead of building the menu in your activity's code, you can define the menu and all its items in an XML menu resource. A menu resource defines an application menu (options menu, context menu, or popup menu) that can be inflated with MenuInflater, which loads the resource as a Menu object in your activity or fragment.

If you start an app project using the Basic Activity template, the template adds the menu resource for you and inflates the options menu with MenuInflater, so you can skip this step and go right to "Defining how menu items appear".

If you are *not* using the Basic Activity template, use the resource-inflate design pattern, which makes it easy to create an options menu. Follow these steps (refer to the figure below):



1. **XML menu resource**. Create an XML menu resource file for the menu items, and assign appearance and position attributes as described in the next section.
2. **Inflating the menu**. Override the onCreateOptionsMenu() method in your activity or fragment to inflate the menu.
3. **Handling menu item clicks**. Menu items are views, so you can use the `android:onClick` attribute for each menu item. However, the onOptionsItemSelected() method can handle all the menu item clicks in one place, and determine which menu item was clicked, which makes your code easier to understand.
4. **Performing actions**. Create a method to perform an action for each options menu item.

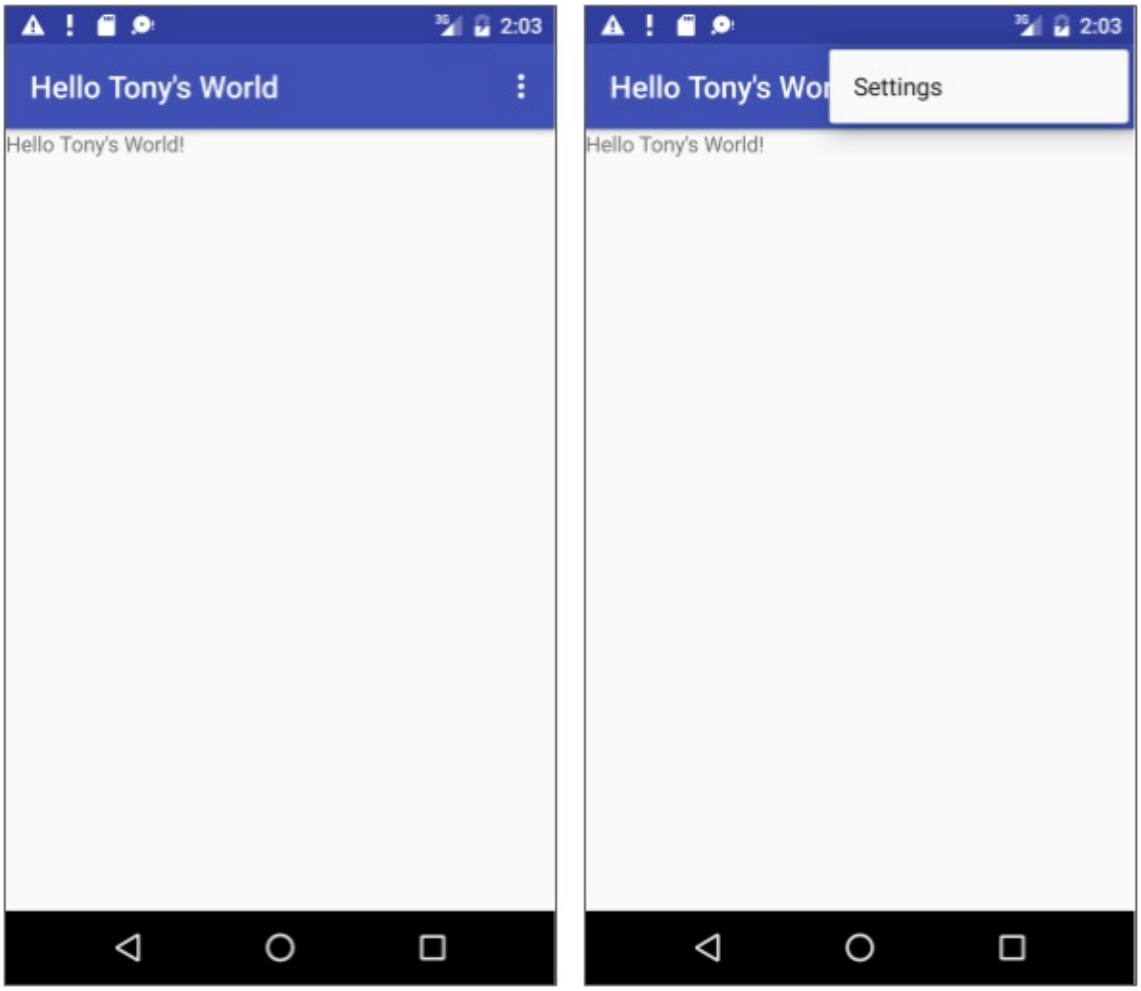
Creating an XML resource for the menu

Follow these steps to add the menu items to an XML menu resource:

1. Click the **res** directory, and choose **File > New > Android resource directory**, choose **menu** in the Resource type drop-down menu, and click **OK**.
2. Click the new **menu** directory, and choose **File > New > Menu resource file**, enter the name of the file as **menu_main** or something similar, and click **OK**. The new menu_main.xml file now resides within the menu directory.
3. Open the **menu_main.xml** file (if not already open), and click the **Text** tab next to the Design tab at the bottom of the pane to show the text of the file.
4. Add the first options menu item using the `<item … />` tag. In this example, the item is **Settings**:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
   ...>
   <item
      android:id="@+id/action_settings"
      android:title="@string/settings" />
</menu>
```

After setting up and inflating the XML resource in the activity or fragment code, the overflow icon in the app bar, when clicked, would show the options menu with just one option (**Settings**):



Defining how menu items appear

If you start an app project using the Basic Activity template, the template adds the options menu with one option: **Settings**.

To add more options menu items, add more `<item … />` tags in the **menu_main.xml** file. For example, in the following snippet, two options menu items are defined: `@string/settings` (**Settings**) and `@string/action_order` (**Order**):

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
   ...>
   <item
      android:id="@+id/action_settings"
      android:title="@string/settings" />
   <item
      android:id="@+id/action_order"
      android:icon="@drawable/ic_order_white"
      android:title="@string/action_order"/>
</menu>
```

Within each `<item … />` tag you can add the attributes to define how the menu item appears, such as the order of its appearance relative to the other items, and whether the item can appear as an icon in the app bar. Any item you set to *not* appear in the app bar (or that can't fit in the app bar given the display orientation) is placed in order in the overflow menu).

Whenever possible, you want to show the most frequently used actions using icons in the app bar so that the user can click them without having to first click the overflow button.

Adding icons for menu items

To specify icons for actions, you need to first add the icons as image assets to the **drawable** folder by following these steps (see Image Asset Studio for a complete description):

1. Expand **res** in the Project view, and right-click (or Command-click) **drawable**.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu.
4. Edit the name of the icon (for example, **ic_order_white** for the Order menu item).
5. Click the clipart image (the Android logo) to select a clipart image as the icon. A page of icons appears. Click the icon you want to use.
6. (Optional) Choose **HOLO_DARK** from the Theme drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next**.
7. Click **Finish** in the Confirm Icon Path dialog.

Icon and appearance attributes

Use the following attributes to govern the menu item's appearance:

- `android:icon` : An image to use as the menu item icon. For example, the following menu item defines `ic_order_white` as its icon:

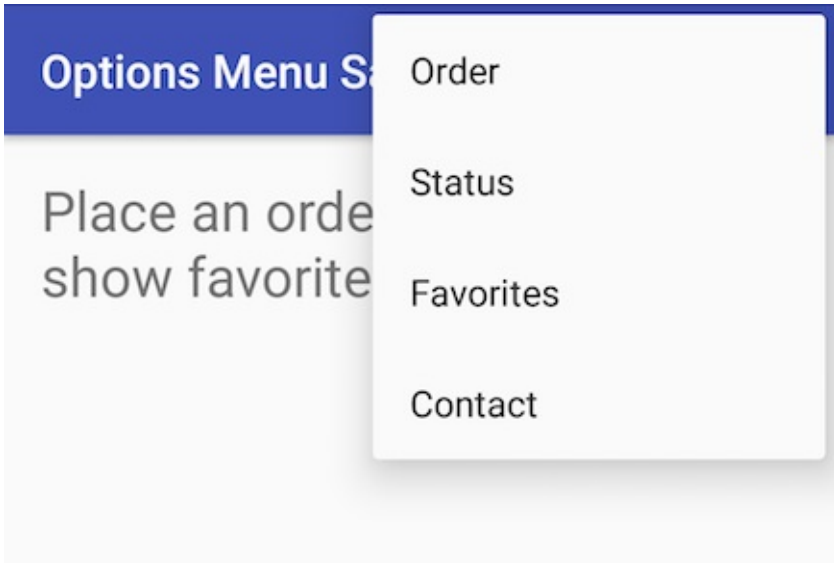
```
<item
    android:id="@+id/action_order"
    android:icon="@drawable/ic_order_white"
    android:title="@string/action_order"/>
```

- `android:title` : A string for the title of the menu item.
- `android:titleCondensed` : A string to use as a condensed title for situations in which the normal title is too long.

Position attributes

Use the `android:orderInCategory` attribute to specify the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu. This is usually the order of importance of the item within the menu. For example, if you want **Order** to be first, followed by **Status**, **Favorites**, and **Contact**, the following table shows the priority of these items in the menu:

Menu Item	`orderInCategory attribute`
Order	10
Status	20
Favorites	40
Contact	100

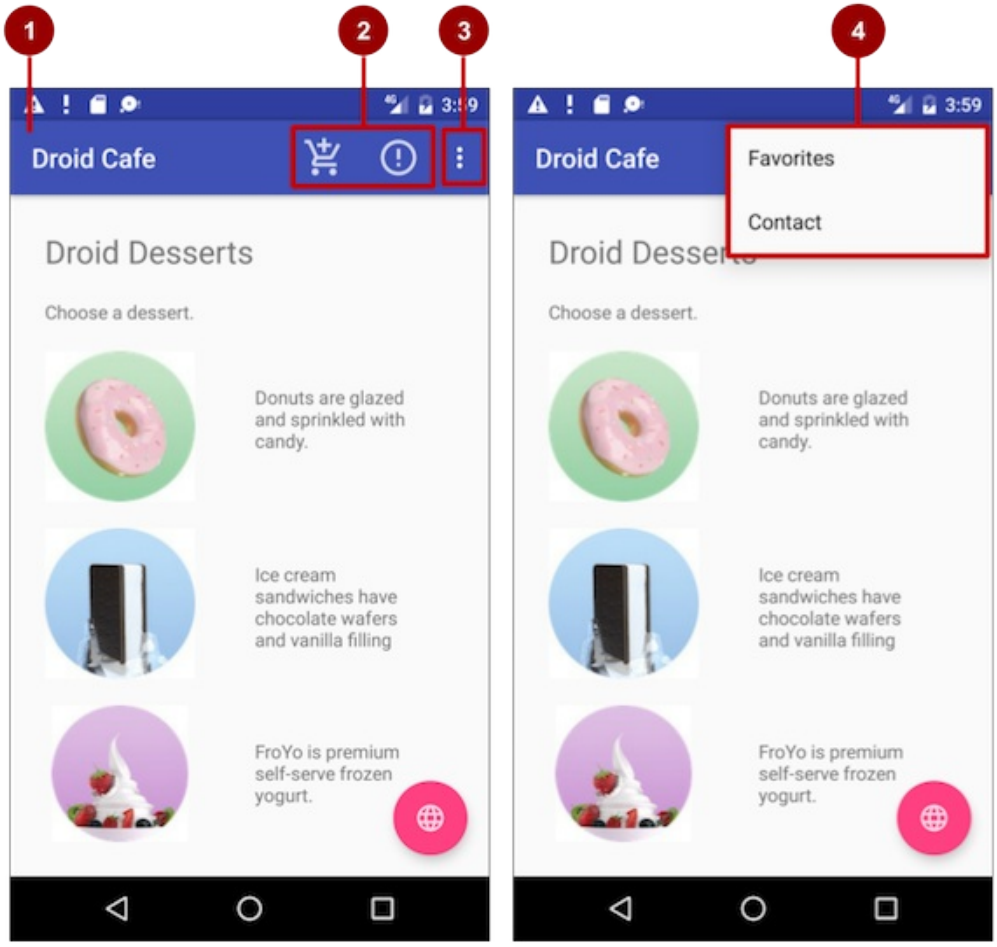


Note: While the numbers 1, 2, 3, and 4 would also work in the above example, the numbers 10, 20, 40, and 100 leave room for additional menu items to be added later between them.
Use the `app:showAsAction` attribute to show menu items as icons in the app bar, with the following values:

- `"always"` : Always place this item in the app bar. Use this only if it's critical that the item appear in the app bar (such as a Search icon). If you set multiple items to always appear in the app bar, they might overlap something else in the app bar, such as the app title.
- `"ifRoom"` : Only place this item in the app bar if there is room for it. If there is not enough room for all the items marked `"ifRoom"` , the items with the lowest `orderInCategory` values are displayed in the app bar, and the remaining items are displayed in the overflow menu.
- `"never"` : Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
- `"withText"` : Also include the title text (defined by `android:title`) with the item. The title text appears anyway if the item appears in the overflow menu, so this attribute is used primarily to include the title with the icon in the app bar.

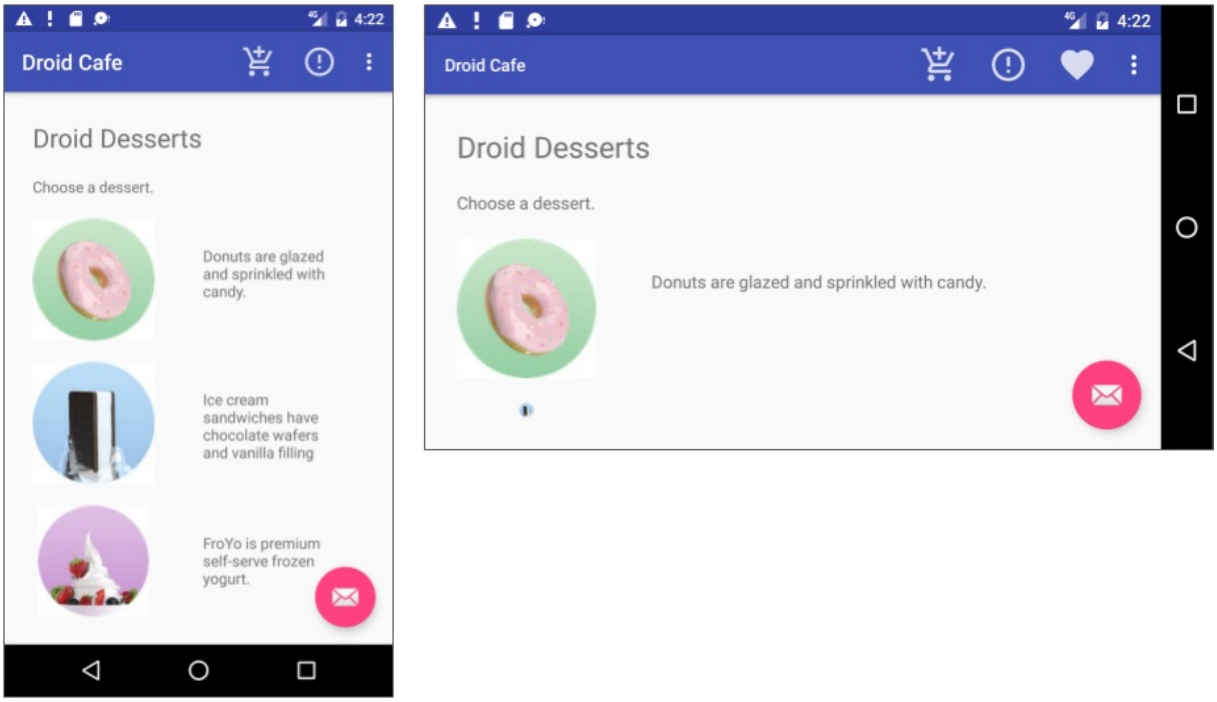
For example, the following menu item's icon appears in the app bar only if there is room for it:

```
<item
   android:id="@+id/action_favorites"
   android:icon="@drawable/ic_favorites_white"
   android:orderInCategory="40"
   android:title="@string/action_favorites"
   app:showAsAction="ifRoom" />
```



In the above figure:

1. **Options menu action icons**. The first two options menu items appear as action icons in the app bar: **Order** (the shopping cart icon) and **Info** (the "**i**" icon).
2. **Overflow button**. Clicking the overflow button shows the overflow menu.
3. **Options overflow menu**. The overflow menu shows more of the options menu: **Favorites** and **Contact**. **Favorites** (the heart icon) doesn't fit into the app bar in vertical orientation, but may appear in horizontal orientation on a smartphone, or in both orientations on a tablet, as shown below.



Inflating the menu resource

If you start an app project using the Basic Activity template, the template adds the code for inflating the options menu with MenuInflater, so you can skip this step.

If you are *not* using the Basic Activity template, inflate the menu resource in your activity by using the onCreateOptionsMenu() method (with the `Override` annotation) with the getMenuInflater() method of the Activity class.

The `getMenuInflater()` method returns a MenuInflater, which is a class used to instantiate menu XML files into Menu objects. The MenuInflater class provides the inflate() method, which takes as a parameter the resource `id` for an XML layout resource to load (`R.menu.menu_main` in the following example), and the Menu to inflate into (`menu` in the following example):

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
   getMenuInflater().inflate(R.menu.menu_main, menu);
   return true;
}
```

Handling the menu item click

As with a button, the `android:onClick` attribute defines a method to call when this menu item is clicked. You must declare the method in the activity as `public` and accept a `MenuItem` as its only parameter, which indicates the item clicked.

For example, you could define the **Favorites** item in the menu resource file to use the `android:onClick` attribute to call the `onFavoritesClick()` method:

```
<item
        android:id="@+id/action_favorites"
        android:icon="@drawable/ic_favorites_white"
        android:orderInCategory="40"
        android:title="@string/action_favorites"
        app:showAsAction="ifRoom"
        android:onClick="onFavoritesClick" />
```

You would declare the `onFavoritesClick()` method in the activity:

```
public void onFavoritesClick(MenuItem item) {
    // The item parameter indicates which item was clicked.
    // Add code to handle the Favorites click.
}
```

However, the onOptionsItemSelected() method of the Activities class can handle all the menu item clicks in one place, and determine which menu item was clicked, which makes your code easier to understand. The Basic Activity template provides an implementation of the `onOptionsItemSelected()` method with a `switch case` block to call the appropriate method (such as `showOrder`) based on the menu item's `id` , which you can retrieve using the getItemId() method of the Adapter class:

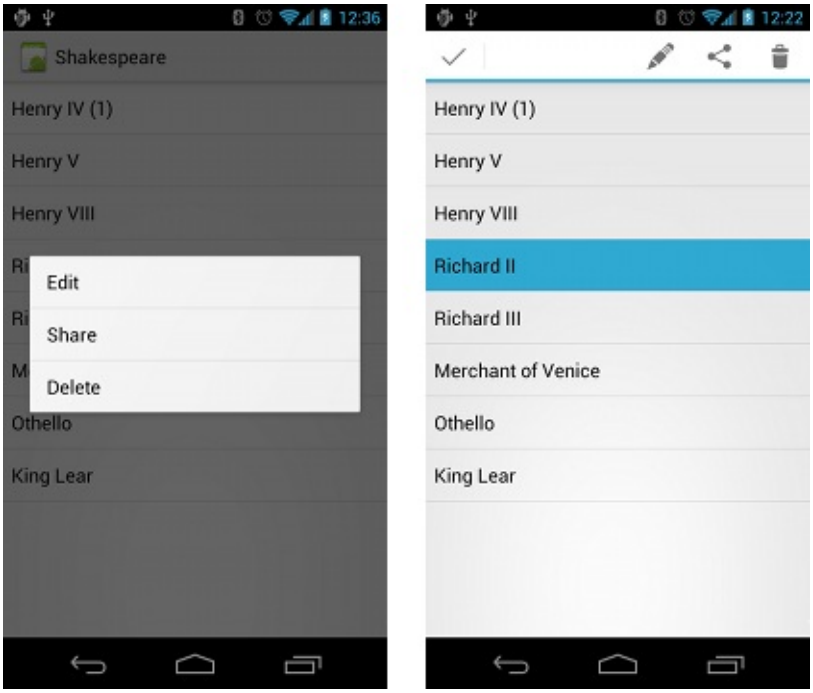
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.action_order:
         showOrder();
         return true;
      case R.id.action_status:
         showStatus();
         return true;
      case R.id.action_contact:
         showContact();
         return true;
      default:
         // Do nothing
   }
   return super.onOptionsItemSelected(item);
}
```

Contextual menu

Use a *contextual menu* to allow users to take an action on a selected view. You can provide a context menu for any View, but they are most often used for items in a RecyclerView, GridView, or other view collections in which the user can perform direct actions on each item.

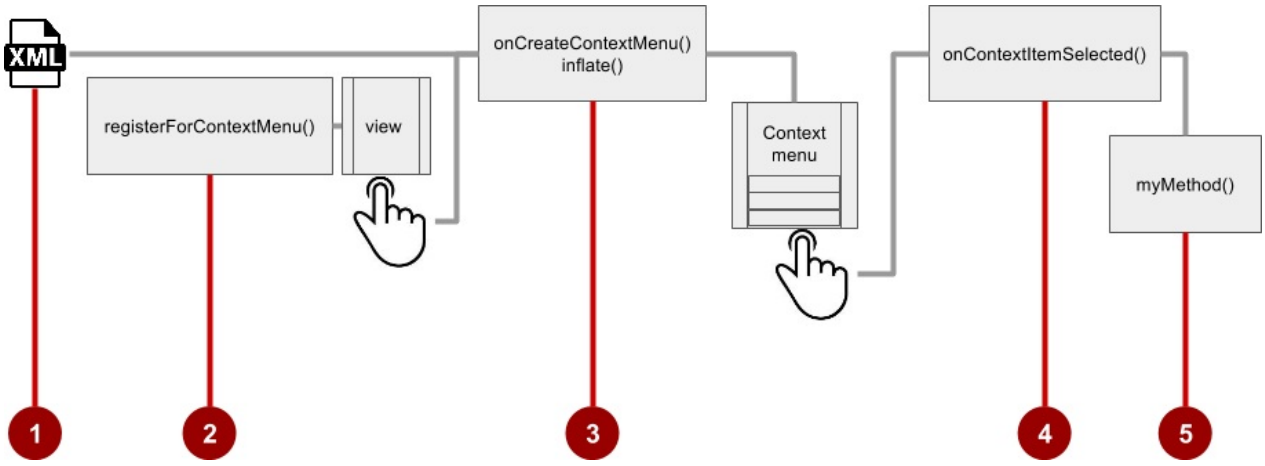
Android provides two kinds of contextual menus:

- A *context menu*, shown on the left side in the figure below, appears as a floating list of menu items when the user performs a long tap on a view element on the screen. It is typically used to modify the view element or use it in some fashion. For example, a context menu might include **Edit** to edit a view element, **Delete** to delete it, and **Share** to share it over social media. Users can perform a contextual action on one view element at a time.
- A *Contextual action bar*, shown on the right side of the figure below, appears at the top of the screen in place of the app bar or underneath the app bar, with action items that affect the selected view element(s). Users can perform an action on multiple view elements at once (if your app allows it).



Floating context menu

The familiar resource-inflate design pattern is used to create a floating context menu, modified to include registering (associating) the context menu with a view:



Follow these steps to create a floating context menu for one or more view elements (refer to figure above):

1. Create an XML menu resource file for the menu items, and assign appearance and position attributes (as described in the previous section).
2. Register a view to the context menu using the registerForContextMenu() method of the Activity class.
3. Implement the onCreateContextMenu() method in your activity or fragment to inflate the menu.
4. Implement the onContextItemSelected() method in your activity or fragment to handle menu item clicks.
5. Create a method to perform an action for each context menu item.

Creating the XML resource file

Create the XML menu resource directory and file by following the steps in the previous section. Use a suitable name for the file, such as `menu_context`. Add the context menu items (in this example, the menu items are **Edit**, **Share**, and **Delete**):

```
<item
   android:id="@+id/context_edit"
   android:title="@string/edit"
   android:orderInCategory="10"/>

<item
   android:id="@+id/context_share"
   android:title="@string/share"
   android:orderInCategory="20"/>

<item
   android:id="@+id/context_delete"
   android:title="@string/delete"
   android:orderInCategory="30"/>
```

Registering a view to the context menu

Register a view to the context menu by calling the registerForContextMenu() method and passing it the view. Registering a context menu for a view sets the View.OnCreateContextMenuListener on the view to this activity, so that onCreateContextMenu() will be called when it is time to show the context menu. (You implement `onCreateContextMenu` in the next section.)

For example, in the `onCreate()` method for the activity, add the `registerForContextMenu()` statement:

```
...
// Registering the context menu to the text view of the article.
TextView article_text = (TextView) findViewById(R.id.article);
registerForContextMenu(article_text);
...
```

Multiple views can be registered to the same context menu. If you want each item in a ListView or GridView to provide the same context menu, register all items for a context menu by passing the ListView or GridView to registerForContextMenu().

Implementing the onCreateContextMenu() method

When the registered view receives a long-click event, the system calls the onCreateContextMenu() method, which you can override in your activity or fragment. This is where you define the menu items, usually by inflating a menu resource.

For example:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                        ContextMenu.ContextMenuInfo menuInfo) {
   super.onCreateContextMenu(menu, v, menuInfo);
   MenuInflater inflater = getMenuInflater();
   inflater.inflate(R.menu.menu_context, menu);
}
```

In the above code:

- The `menu` parameter for onCreateContextMenu() is the context menu to be built.
- The `v` parameter is the view registered for the context menu.
- The `menuInfo` parameter is extra information about the view registered for the context menu. This information varies depending on the class of **v**, which could be a RecyclerView or a GridView. If you are registering a RecyclerView or GridView, you would instantiate a ContextMenu.ContextMenuInfo object to provide the information about the item selected, and pass it as `menuInfo`, such as the row id, position, or child view.

The MenuInflater class provides the inflate() method, which takes as a parameter the resource id for an XML layout resource to load (`menu_context` in the above example), and the Menu to inflate into (`menu` in the above example).

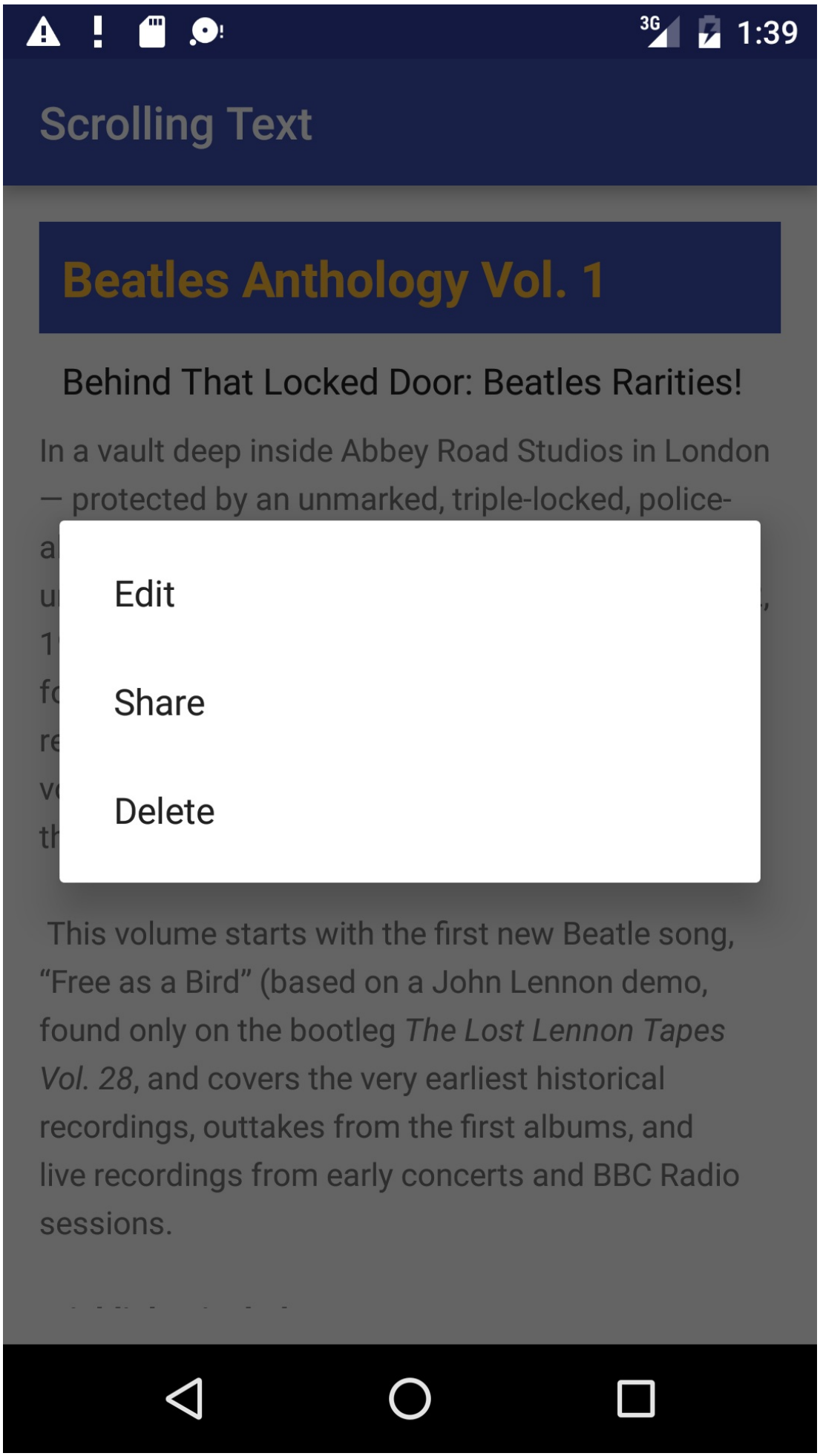
Implementing the onContextItemSelected() method

When the user clicks on a menu item, the system calls the onContextItemSelected() method. You override this method in your activity or fragment in order to determine which menu item was clicked, and for which view the menu is appearing. You also use it to implement the appropriate action for the menu items, such as `editNote()` and `shareNote()` below for the **Edit** and **Share** menu items. For example:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.context_edit:
         editNote();
         return true;
      case R.id.context_share:
         shareNote();
         return true;
      default:
         return super.onContextItemSelected(item);
   }
}
```

The above example uses the getItemId() method to get the `id` for the selected menu item, and uses it in a `switch case` block to determine which action to take. The `id` is the `android:id` attribute assigned to the menu item in the XML menu resource file.

When the user performs a long-click on the article in the text view, the floating context menu appears and the user can click a menu item.


1:39
Scrolling Text
Beatles Anthology Vol. 1
Behind That Locked Door: Beatles Rarities!
In a vault deep inside Abbey Road Studios in London — protected by an unmarked, triple-locked, police-
Edit
Share
Delete
This volume starts with the first new Beatle song, “Free as a Bird” (based on a John Lennon demo, found only on the bootleg *The Lost Lennon Tapes Vol. 28*, and covers the very earliest historical recordings, outtakes from the first albums, and live recordings from early concerts and BBC Radio sessions.

If you are using the `menuInfo` information for a RecyclerView or GridView, you would add a statement before the switch case block to gathers the specific information about the selected view (for `info`) by using AdapterView.AdapterContextMenuInfo:

```
AdapterView.AdapterContextMenuInfo info =
            (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
```

Contextual action bar

A *contextual action bar* appears at the top of the screen to present actions the user can perform on a view after long-clicking the view, as shown in the figure below.

1:39
1
Beatles Anthology Vol. 1
Behind That Locked Door: Beatles Rarities!
In a vault deep inside Abbey Road Studios in London -- protected by an unmarked, triple-locked, police-alarmed door -- are something like 400 hours of unreleased Beatles recordings, starting from June 2, 1962 and ending with the very last tracks recorded for the Let It Be album. The best of the best were released by Apple Records in the form of the 3-volume Anthology series. For more information, see the Beatles Time Capsule at www.rockument.com.
This volume starts with the first new Beatle song, "Free as a Bird" (based on a John Lennon demo, found only on the bootleg The Lost Lennon Tapes Vol. 28, and covers the very earliest historical recordings, outtakes from the first albums, and live recordings from early concerts and BBC Radio sessions.
2

In the above figure:

1. **Contextual action bar**. The bar offers three actions on the right side (**Edit**, **Share**, and **Delete**) and the **Done** button (left arrow icon) on the left side.
2. **View**. View on which a long-click triggers the contextual action bar.

The contextual action bar appears only when *contextual action mode*, a system implementation of ActionMode, occurs as a result of the user performing a long-click on the View.

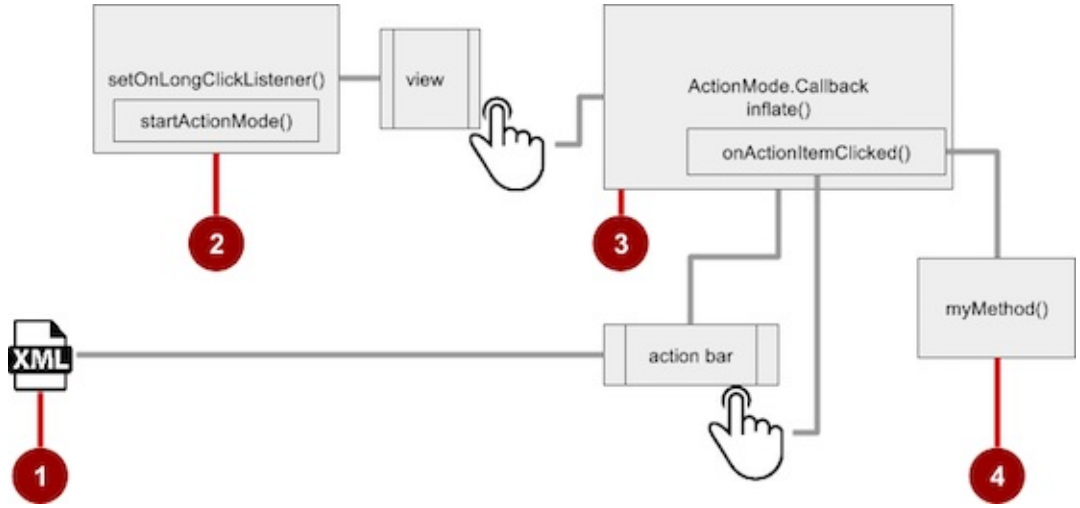
ActionMode represents a user interface (UI) mode for providing alternative interaction, replacing parts of the normal UI until finished. For example, text selection is implemented as an ActionMode, as are contextual actions that work on a selected item on the screen. Selecting a section of text or long-clicking a view triggers ActionMode.

While this mode is enabled, the user can select multiple items (if your app allows it), deselect items, and continue to navigate within the activity. The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the Back button, or taps **Done** (left-arrow icon) on the left side of the bar.

Follow these steps to create a contextual action bar (refer to the figure below):

1. Create an XML menu resource file for the menu items, and assign an icon to each one (as described in a previous section).
2. Set the long-click listener to the view that should trigger the contextual action bar using the setOnLongClickListener() method. Call startActionMode() within the setOnLongClickListener() method when the user performs a long tap on the view.

3. Implement the ActionMode.Callback interface to handle the ActionMode lifecycle. Include in this interface the action for responding to a menu item click in the onActionItemClicked() callback method.
4. Create a method to perform an action for each context menu item.



Creating the XML resource file

Create the XML menu resource directory and file by following the steps in a previous section. Use a suitable name for the file, such as `menu_context`. Add icons for the context menu items (in this example, the menu items are **Edit**, **Share**, and **Delete**). For example, the Edit menu item would have these attributes:

```
<item
   android:id="@+id/context_edit"
   android:orderInCategory="10"
   android:icon="@drawable/ic_action_edit_white"
   android:title="@string/edit" />
```

The standard contextual action bar has a dark background. Use a light or white color for the icons. If you are using clip art icons, choose **HOLO_DARK** for the Theme drop-down menu when creating the new image asset.

Setting the long-click listener

Use setOnLongClickListener() to set a long-click listener to the View that should trigger the contextual action bar. Add the code to set the long-click listener to the activity class (such as **MainActivity**) using the activity's `onCreate()` method. Follow these steps:

1. Declare the member variable `mActionMode` in the class definition for the activity:

   ```
   private ActionMode mActionMode;
   ```

   You will call startActionMode() to enable ActionMode, which returns the ActionMode created. By saving this in a member variable (`mActionMode`), you can make changes to the contextual action bar in response to other events.

2. Set up the contextual action bar listener in the `onCreate()` method, using `View` as the type for the view in order to use the `setOnLongClickListener` :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   ...
   View articleView = findViewById(article);
   articleView.setOnLongClickListener(new View.OnLongClickListener() {
      ...
      // Add method here to start ActionMode after long-click.
      ...
   });
}
```

Implementing the ActionMode.Callback interface

Before you can add the code to `onCreate()` to start ActionMode, you must implement the ActionMode.Callback interface to manage the action mode lifecycle. In its callback methods, you can specify the actions for the contextual action bar, and respond to clicks on action items.

1. Add the following method to the activity class (such as **MainActivity**) to implement the interface:

   ```
   public ActionMode.Callback mActionModeCallback = new
                                      ActionMode.Callback() {
      ...
      // Add code to create action mode here.
      ...
   }
   ```

2. Add the `onCreateActionMode()` code within the brackets of the above method to create action mode (the full code is provided at the end of this section):

   ```
   @Override
   public boolean onCreateActionMode(ActionMode mode, Menu menu) {
     // Inflate a menu resource providing context menu items
     MenuInflater inflater = mode.getMenuInflater();
     inflater.inflate(R.menu.menu_context, menu);
     return true;
   }
   ```

   The onCreateActionMode() method inflates the menu using the same pattern used for a floating context menu. But this inflation occurs *only* when ActionMode is created, which is when the user performs a long-click. The MenuInflater class provides the inflate() method, which takes as a parameter the resource `id` for an XML layout resource to load (`menu_context` in the above example), and the Menu to inflate into (`menu` in the above example).

3. Add the onActionItemClicked() method with your handlers for each menu item:

   ```
   @Override
   public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
   switch (item.getItemId()) {
     case R.id.context_edit:
        editNote();
        mode.finish();
        return true;
     case R.id.context_share:
        shareNote();
        mode.finish();
        return true;
     default:
        return false;
   }
   ```

   The above code above uses the getItemId() method to get the `id` for the selected menu item, and uses it in a `switch` `case` block to determine which action to take. The `id` in each `case` statement is the `android:id` attribute assigned to the menu item in the XML menu resource file.

The actions shown are the `editNote()` and `shareNote()` methods, which you can create in the same activity. After the action is picked, you use the `mode.finish()` method to close the contextual action bar.

4. Add the onPrepareActionMode() and onDestroyActionMode() methods, which manage the ActionMode lifecycle:

```
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
      return false; // Return false if nothing is done.
}
```

The `onPrepareActionMode()` method shown above is called each time ActionMode occurs, and is always called after `onCreateActionMode()`.

```
@Override
public void onDestroyActionMode(ActionMode mode) {
      mActionMode = null;
}
```

The `onDestroyActionMode()` method shown above is called when the user exits ActionMode by clicking **Done** in the contextual action bar, or clicking on a different view.

5. Review the full code for the ActionMode.Callback interface implementation:

```
public ActionMode.Callback mActionModeCallback = new
                                        ActionMode.Callback() {
   @Override
   public boolean onCreateActionMode(ActionMode mode, Menu menu) {
      // Inflate a menu resource providing context menu items
      MenuInflater inflater = mode.getMenuInflater();
      inflater.inflate(R.menu.menu_context, menu);
      return true;
   }

   // Called each time ActionMode is shown. Always called after
   // onCreateActionMode.
   @Override
   public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
      return false; // Return false if nothing is done
   }

   // Called when the user selects a contextual menu item
   @Override
   public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
      switch (item.getItemId()) {
         case R.id.context_edit:
            editNote();
            mode.finish();
            return true;
         case R.id.context_share:
            shareNote();
            mode.finish();
            return true;
         default:
            return false;
      }
   }

   // Called when the user exits the action mode
   @Override
   public void onDestroyActionMode(ActionMode mode) {
      mActionMode = null;
   }
};
```

Starting ActionMode

You use startActionMode() to start ActionMode after the user performs a long-click.

1. To start ActionMode, add the `onLongClick()` method within the brackets of the `setOnLongClickListener` method in `onCreate()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   ...
   articleView.setOnLongClickListener(new View.OnLongClickListener() {
      // Called when the user long-clicks on articleView
      public boolean onLongClick(View view) {
         if (mActionMode != null) return false;
         // Start the contextual action bar
         // using the ActionMode.Callback.
         mActionMode =
               MainActivity.this.startActionMode(mActionModeCallback);
         view.setSelected(true);
         return true;
      }
   });
}
```

   The above code first ensures that the `ActionMode` instance is not recreated if it's already active by checking whether `mActionMode` is null before starting the action mode:

```
if (mActionMode != null) return false;
```

   When the user performs a long-click, the call is made to `startActionMode()` using the ActionMode.Callback interface, and the contextual action bar appears at the top of the display. The setSelected() method changes the state of this view to selected (set to `true`).

2. Review the code for the `onCreate()` method in the activity, which now includes `setOnLongClickListener()` and `startActionMode()`:

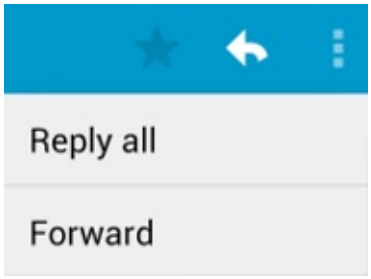
```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   // set up the contextual action bar listener
   View articleView = findViewById(article);
   articleView.setOnLongClickListener(new View.OnLongClickListener() {
      // Called when the user long-clicks on articleView
      public boolean onLongClick(View view) {
         if (mActionMode != null) return false;
         // Start the contextual action bar
         // using the ActionMode.Callback.
         mActionMode =
               MainActivity.this.startActionMode(mActionModeCallback);
         view.setSelected(true);
         return true;
      }
   });
}
```

Popup menu

A PopupMenu is a vertical list of items anchored to a View. It appears below the anchor view if there is room, or above the view otherwise.

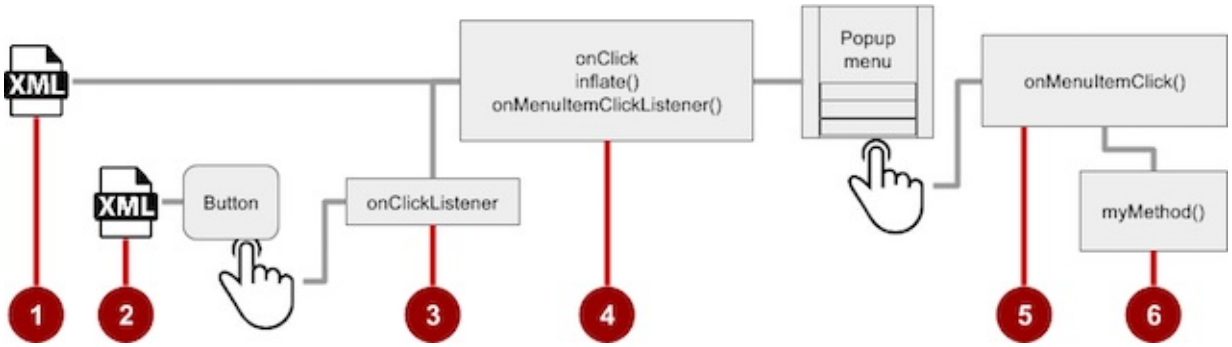
A popup menu is typically used to provide an overflow of actions (similar to the overflow action icon for the options menu) or the second part of a two-part command. Use a popup menu for extended actions that relate to regions of content in your activity. Unlike a context menu, a popup menu is anchored to a Button (View), is always available, and it's actions generally do not affect the content of the View.

For example, the Gmail app uses a popup menu anchored to the overflow icon in the app bar when showing an email message. The popup menu items **Reply**, **Reply All**, and **Forward** are *related* to the email message, but don't *affect* or *act on* the message. Actions in a popup menu should not directly affect the corresponding content (use a contextual menu to directly affect selected content). As shown below, a popup can be anchored to the overflow action button in the action bar.



Creating a pop-up menu

Follow these steps to create a popup menu (refer to figure below):



1. Create an XML menu resource file for the popup menu items, and assign appearance and position attributes (as described in a previous section).
2. Add an ImageButton for the popup menu icon in the XML activity layout file.
3. Assign onClickListener() to the button.
4. Override the `onClick()` method to inflate the popup menu and register it with PopupMenu.OnMenuItemClickListener.
5. Implement the onMenuItemClick() method.
6. Create a method to perform an action for each popup menu item.

Creating the XML resource file

Create the XML menu resource directory and file by following the steps in a previous section. Use a suitable name for the file, such as `menu_popup` .

Adding an ImageButton for the icon to click

Use an ImageButton in the activity layout for the icon that triggers the popup menu. Popup menus are anchored to a view in the activity, such as an ImageButton. The user clicks it to see the menu.

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button_popup"
    android:src="@drawable/@drawable/ic_action_popup"/>
```

Assigning onClickListener to the button

1. Create a member variable (`mButton`) in the activity's class definition:

```
public class MainActivity extends AppCompatActivity {
   private ImageButton mButton;
   ...
}
```

2. In the `onCreate()` method for the same activity, assign the `ImageButton` in the layout to the member variable, and assign onClickListener() to the button:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   ...
   mButton = (ImageButton) findViewById(R.id.button_popup);
   mButton.setOnClickListener(new View.OnClickListener() {
   ...
   // define onClick here
   ...
});
```

Inflating the popup menu

As part of the `setOnClickListener()` method within `onCreate()` , add the `onClick()` method to inflate the popup menu and register it with PopupMenu.OnMenuItemClickListener:

```
@Override
public void onClick(View v) {
   //Creating the instance of PopupMenu
   PopupMenu popup = new PopupMenu(MainActivity.this, mButton);
   //Inflating the Popup using xml file
   popup.getMenuInflater().inflate(R.menu.menu_popup, popup.getMenu());
   //registering popup with OnMenuItemClickListener
   popup.setOnMenuItemClickListener(new
                            PopupMenu.OnMenuItemClickListener() {
      ...
      // Add onMenuItemClick here
      ...
            // Perform action here
      ...
}
```

After instantiating a PopupMenu object (`popup` in the above example), the method uses the MenuInflater class and its inflate() method, which takes as parameters:

- The resource `id` for an XML layout resource to load (`menu_popup` in the example above)
- The Menu to inflate into (`popup.getMenu()` in the example above).

The code then registers the popup with the listener, PopupMenu.OnMenuItemClickListener.

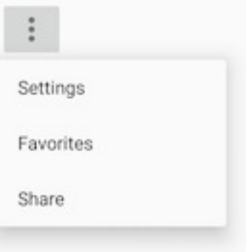
Implementing onMenuItemClick

To perform an action when the user selects a popup menu item, implement the onMenuItemClick() callback within the above `setOnClickListener()` method, and finish the method with `popup.show` to show the popup menu:

```
         public boolean onMenuItemClick(MenuItem item) {
            // Perform action here
            return true;
         }
      });
      popup.show();  //show the popup menu
   }
});// close the setOnClickListener method
```

Putting these pieces together, the entire `onCreate()` method should now look like this:

```
private ImageButton mButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   ...
   // popup button setup
   mButton = (ImageButton) findViewById(R.id.button_popup);
   mButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         //Creating the instance of PopupMenu
         PopupMenu popup = new PopupMenu(MainActivity.this, mButton);
         //Inflating the Popup using xml file
         popup.getMenuInflater().inflate(R.menu.menu_popup, popup.getMenu());

         //registering popup with OnMenuItemClickListener
         popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
            public boolean onMenuItemClick(MenuItem item) {
               // Perform action here
               return true;
            }
         });
         popup.show();//show the popup menu
      }
   });//close the setOnClickListener method
 }
```



Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Using an Options Menu

Learn more

- Android API Guide, "Develop" section:
  - Adding the App Bar
  - Styles and Themes
  - Menus

    - Menu Resource
- Other:
    - Android API Guide, "Design" section: Icons and other downloadable resources
    - Android Studio User's Guide: Image Asset Studio
    - Android Developers Blog: "Holo Everywhere"

4.3: Screen Navigation

Contents:

- Providing users with a path through your app
- Back-button navigation
- Hierarchical navigation patterns
- Ancestral navigation (the Up button)
- Descendant navigation
- Lateral navigation with tabs and swipes
- Related practical
- Learn more

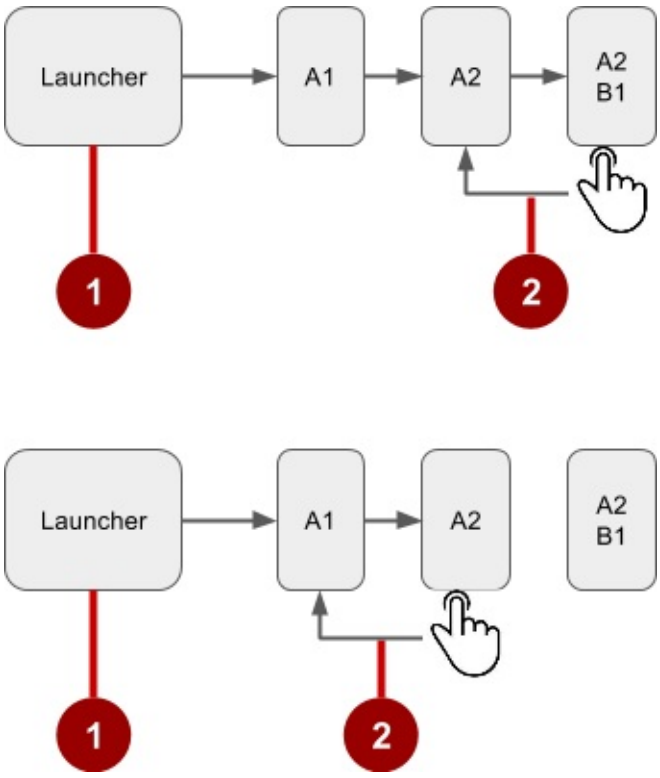
Providing users with a path through your app

In the early stages of developing an app, you should determine the paths users should take through your app in order to do something, such as placing an order or browsing through content. Each path enables users to navigate across, into, and back out from the different tasks and pieces of content within the app.

In many cases you will need several different paths through your app that offer the following types of navigation:

- *Back* navigation: Users can navigate back to the previous screen using the Back button.
- *Hierarchical* navigation: Users can navigate through a hierarchy of screens organized with a *parent* screen for every set of *child* screens.

Back-button navigation



In the above figure:

1. Starting from Launcher.
2. Clicking the Back button to navigate to the previous screen.

You don't have to manage the Back button in your app. The system handles tasks and the *back stack*—the list of previous screens—automatically. The Back button by default simply traverses this list of screens, removing the current screen from the list as the user presses it.

There are, however, cases where you may want to override the behavior for the Back button. For example, if your screen contains an embedded web browser in which users can interact with page elements to navigate between web pages, you may wish to trigger the embedded browser's default back behavior when users press the device's Back button.

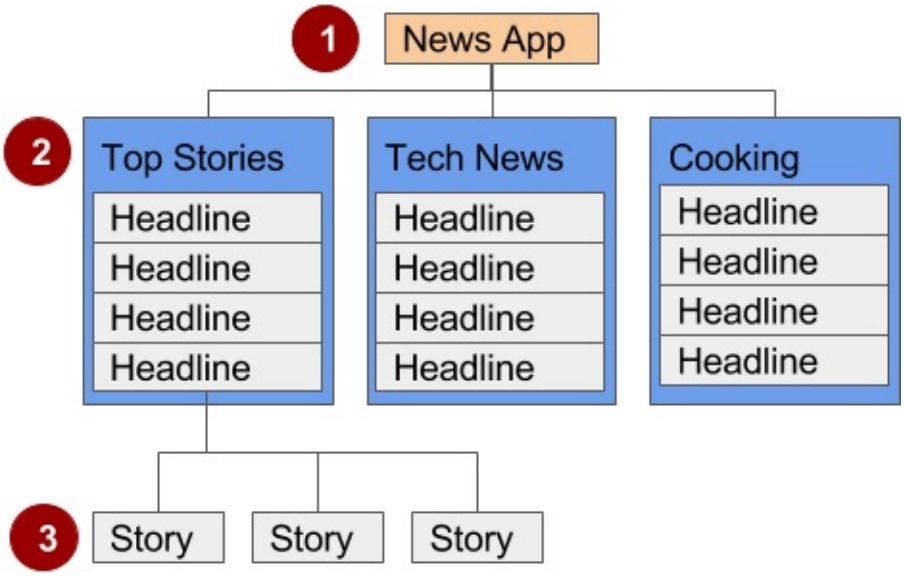
The onBackPressed() method of the Activity class is called whenever the activity detects the user's press of the Back key. The default implementation simply finishes the current activity, but you can override this to do something else:

```
@Override
public void onBackPressed() {
    // Add the Back key handler here.
    return;
}
```

If your code triggers an embedded browser with its own behavior for the Back key, you should return the Back key behavior to the system's default behavior if the user uses the Back key to go beyond the beginning of the browser's internal history.

Hierarchical navigation patterns

To give the user a path through the full range of an app's screens, the best practice is to use some form of hierarchical navigation. An app's screens are typically organized in a parent-child hierarchy, as shown in the figure below:



In the figure above:

1. **Parent screen**. A parent screen (such as a news app's home screen) enables navigation down to *child* screens.
    - The main activity of an app is usually the parent screen.
    - Implement a parent screen as an Activity with *descendant* navigation to one or more child screens.
2. **First-level child screen siblings**. Siblings are screens in the same position in the hierarchy that share the same parent screen (like brothers and sisters).
    - In the first level of siblings, the child screens may be *collection* screens that collect the headlines of stories, as shown above.
    - Implement each child screen as an Activity or Fragment.
    - Implement *lateral* navigation to navigate from one sibling to another on the same level.
    - If there is a second level of screens, the first level child screen is the *parent* to the second level child screen siblings. Implement *descendant* navigation to the second-level child screens.
3. **Second-level child screen siblings**. In news apps and others that offer multiple levels of information, the second level of child screen siblings might offer content, such as stories.
    - Implement a second-level child screen sibling as another Activity or Fragment.
    - Stories at this level may include embedded story elements such as videos, maps, and comments, which might be implemented as fragments.

You can enable the user to navigate up to and down from a parent, and sideways among siblings:

- *Descendant* navigation: Navigating down from a parent screen to a child screen.
- *Ancestral* navigation: Navigating up from a child screen to a parent screen.
- *Lateral* navigation: Navigating from one sibling to another sibling (at the same level).

You can use a main activity (as a parent screen) and then other activities or fragments to implement a hierarchy of screens within an app.

Main activity with other activities

If the first-level child screen siblings have another level of child screens under them, you should implement the first-level screens as activities, so that their lifecycles are managed properly before calling any second-level child screens.

For example, in the figure above, the parent screen is most likely the main activity. An app's main activity (usually MainActivity.java) is typically the parent screen for all other screens in your app, and you implement a navigation pattern in the main activity to enable the user to go to other activities or fragments. For example, you can implement navigation using an Intent that starts an activity.

Tip: Using an Intent in the current activity to start another activity adds the current activity to the call stack, so that the **Back** button in the other activity (described in the previous section) returns the user to the current activity.

As you learned previously, the Android system initiates code in an Activity instance with callback methods that manage the activity's lifecycle for you. (A previous lesson covers the activity lifecycle; for more information, see "Managing the Activity Lifecycle" in the Training section of the Android Developer Develop guide.)

The hierarchy of parent and child activities is defined in the AndroidManifest.xml file. For example, the following defines `OrderActivity` as a child of the parent `MainActivity`:

```
<activity android:name=".OrderActivity"
   android:label="@string/title_activity_order"
   android:parentActivityName=
                       "com.example.android.droidcafe.MainActivity">
   <meta-data
      android:name="android.support.PARENT_ACTIVITY"
      android:value=".MainActivity"/>
</activity>
```

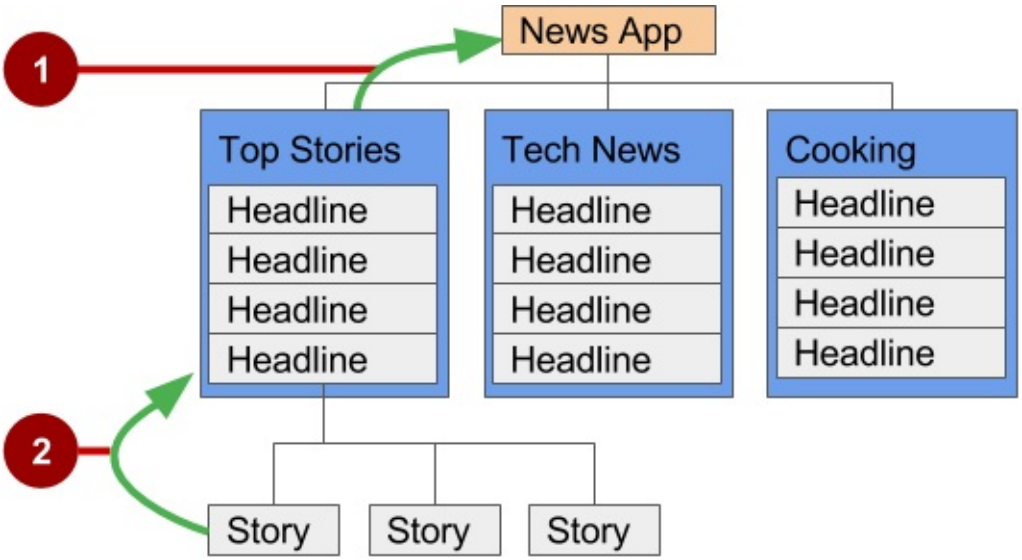
Main activity with fragments

If the child screen siblings do *not* have another level of child screens under them, you can implement them as fragments. A Fragment represents a behavior or portion of a user interface within in an activity. Think of a fragment as a modular section of an activity which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running.

You can combine multiple fragments in a single activity. For example, in a section sibling screen showing a news story and implemented as an activity, you might have a child screen for a video clip implemented as a fragment. You would implement a way for the user to navigate to the video clip fragment, and then back to the activity showing the story.

Ancestral navigation (the Up button)

With ancestral navigation in a multitier hierarchy, you enable the user to go *up* from a section sibling to the collection sibling, and then *up* to the parent screen.



In the above figure:

1. **Up** button for ancestral navigation from the first-level siblings to the parent.
2. **Up** button for ancestral navigation from second-level siblings to the first-level child screen acting as a parent screen.

The **Up** button is used to navigate within an app based on the hierarchical relationships between screens. For example (referring to the figure above):

- If a first-level child screen offers headlines to navigate to second-level child screens, the second-level child screen siblings should offer **Up** buttons that return to the first-level child screen, which is their shared *parent*.
- If the parent screen offers navigation to first-level child siblings, then the first-level child siblings should offer an **Up** button that returns to the parent screen.
- If the parent screen is the topmost screen in an app (that is, the app's home screen), it should not offer an **Up** button.

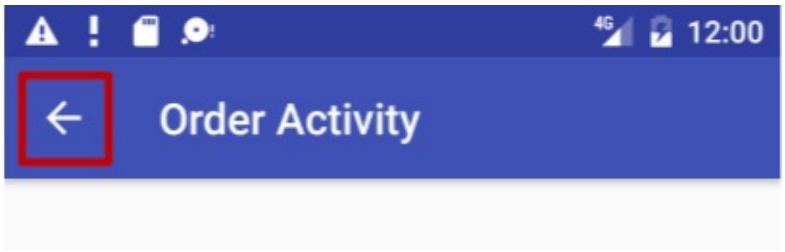
Tip: The **Back** button below the screen differs from the **Up** button. The **Back** button provides navigation to whatever screen you viewed previously. If you have several children screens that the user can navigate through using a lateral navigation pattern (as described later in this chapter), the **Back** button would send the user back to the previous child screen, not to the parent screen. Use an **Up** button if you want to provide ancestral navigation from a child screen back to the parent screen. For more information about Up navigation, see Providing Up Navigation.

See the "Menus" concept chapter for details on how to implement the app bar. To provide the **Up** button for a child screen activity, declare the activity's parent to be MainActivity in the AndroidManifest.xml file. You can also set the `android:label` to a title for the activity screen, such as "Order Activity" (extracted into the string resource `title_activity_order` in the code below). Follow these steps to declare the parent in AndroidManifest.xml:

1. Open **AndroidManifest.xml**.
2. Change the activity element for the child screen activity (in this example, `OrderActivity`) to the following:

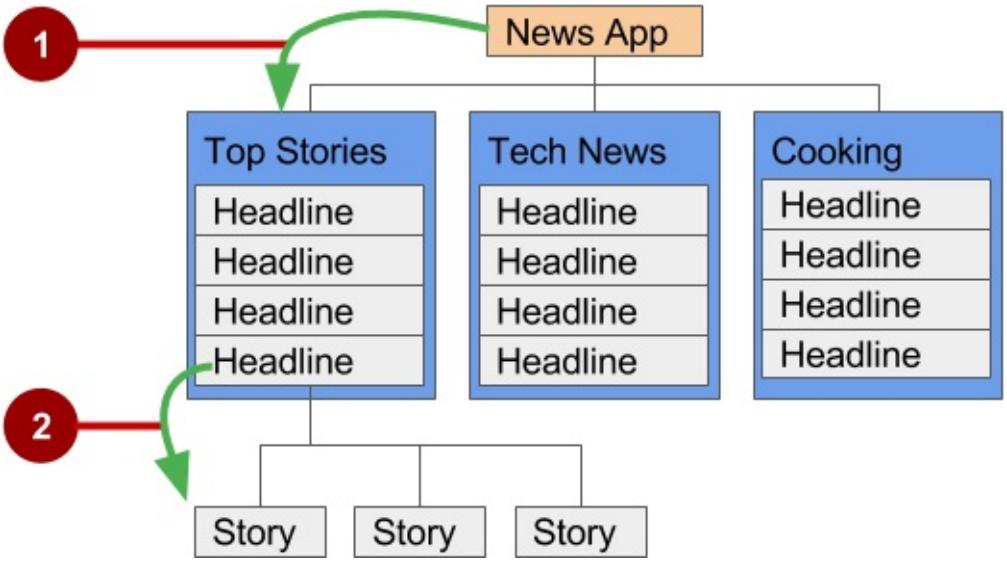
```
 <activity android:name=".OrderActivity"
   android:label="@string/title_activity_order"
   android:parentActivityName=
           "com.example.android.optionsmenuorderactivity.MainActivity">
   <meta-data
      android:name="android.support.PARENT_ACTIVITY"
      android:value=".MainActivity"/>
</activity>
```

The child ("Order Activity") screen now includes the **Up** button in the app bar (highlighted in the figure below), which the user can tap to navigate back to the parent screen.



Descendant navigation

With descendant navigation, you enable the user to go from the parent screen to a first-level child screen, and from a first-level child screen down to a second-level child screen.



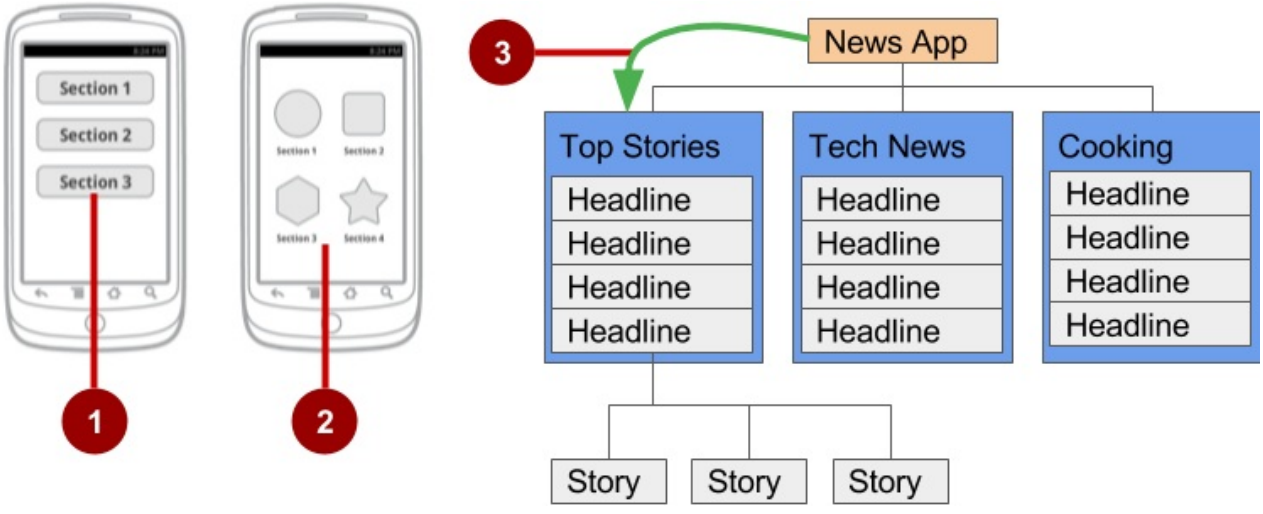
In the above figure:

1. Descendant navigation from parent to first-level child screen.
2. Descendant navigation from headline in a first-level child screen to a second-level child screen.

Buttons or targets

The best practice for descendant navigation from the parent screen to collection siblings is to use buttons or simple *targets* such as an arrangement of images or iconic buttons (also known as a *dashboard*). When the user touches a button, the collection sibling screen opens, replacing the current context (screen) entirely.

Tip: Buttons and simple targets are rarely used for navigating to section siblings *within* a collection. See lists, carousels, and cards in the next section.



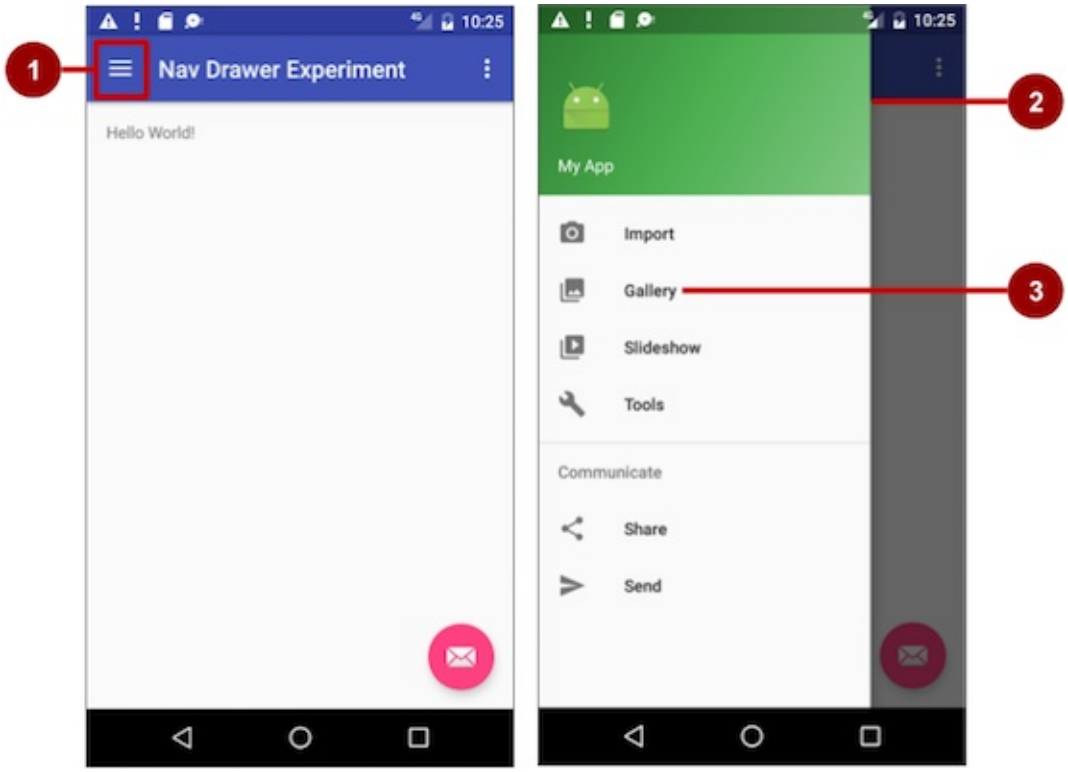
In the figure above:

1. Buttons on a parent screen.
2. Targets (Image buttons or icons) on a parent screen.
3. Descendant navigation pattern from parent screen to first-level child siblings.

A dashboard usually has either two or three rows and columns, with large touch targets to make it easy to use. Dashboards are best when each collection sibling is equally important. You can use a LinearLayout, RelativeLayout, or GridLayout. See Layouts for an overview of how layouts work.

Navigation drawer

A *navigation drawer* is a panel that usually displays navigation options on the left edge of the screen, as shown on the right side of the figure below. It is hidden most of the time, but is revealed when the user swipes a finger from the left edge of the screen or touches the navigation icon in the app bar, as shown on the left side of the figure below.



In the above figure:

1. Navigation icon in the app bar
2. Navigation drawer
3. Navigation drawer menu item

A good example of a navigation drawer is in the Gmail app, which provides access to the Inbox, labelled email folders, and settings. The best practice for employing a navigation drawer is to provide descendant navigation from the parent activity to all of the other activities or fragments in an app. It can display many navigation targets at once—for example, it could contain buttons (like a dashboard), tabs, or a list of items (like the Gmail drawer).

To make a navigation drawer in your app, you need to do the following:

1. Create the following layouts:
    - A navigation drawer as the activity layout's root view.
    - A navigation view for the drawer itself.
    - An app bar layout that will include a navigation icon button.
    - A content layout for the activity that displays the navigation drawer.
    - A layout for the navigation drawer header.
2. Populate the navigation drawer menu with item titles and icons.
3. Set up the navigation drawer and item listeners in the activity code.
4. Handle the navigation menu item selections.

Creating the navigation drawer layout

To create a navigation drawer layout, use the DrawerLayout APIs available in the Support Library. For design specifications, follow the design principles for navigation drawers in the Navigation Drawer design guide.

To add a navigation drawer, use a `DrawerLayout` as the root view of your activity's layout. Inside the `DrawerLayout`, add one view that contains the main content for the screen (your primary layout when the drawer is hidden) and another view, typically a NavigationView, that contains the contents of the navigation drawer.

Tip: To make your layouts simpler to understand, use the `include` tag to include an XML layout within another XML layout.

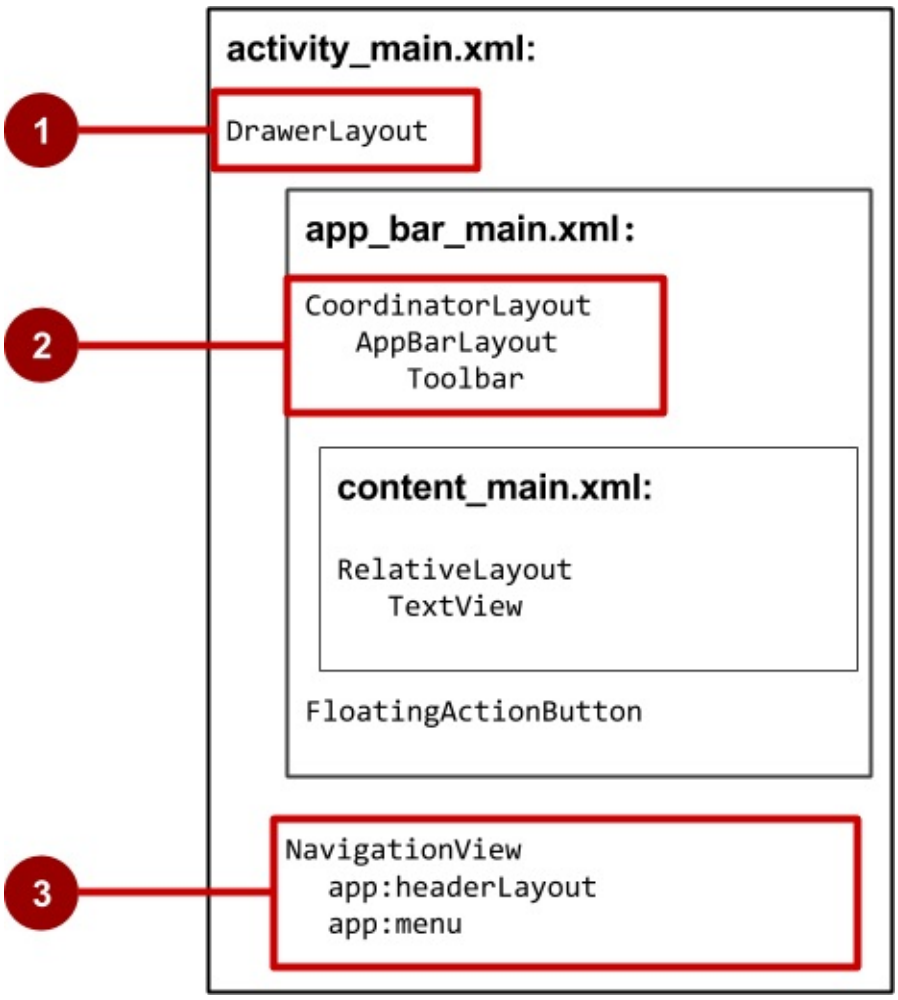
For example, the following layout uses:

- A `DrawerLayout` as the root of the activity's layout in **activity_main.xml**.
- The main content of screen defined in the **app_bar_main.xml** layout file.
- A NavigationView that represents a standard navigation menu that can be populated by a menu resource XML file.

Refer to the figure below that corresponds to this layout:

activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</android.support.v4.widget.DrawerLayout>
```



In the above figure:

1. DrawerLayout is the root view of the activity's layout.
2. The included `app_bar_main` (**app_bar_main.xml**) uses a CoordinatorLayout as its root, and defines the app bar layout with a Toolbar which will include the navigation icon to open the drawer.
3. The NavigationView defines the navigation drawer layout and its header, and adds menu items to it.

Note the following in the **activity_main.xml** layout:

- The `android:id` for the DrawerLayout view is `drawer_layout`. You will use this id to instantiate a `drawer` object in your code.
- The `android:id` for the `NavigationView` is `nav_view`. You will use this id to instantiate a `navigationView` object in your code.
- The `NavigationView` must specify its horizontal gravity with the `android:layout_gravity` attribute. Use the `"start"` value for this attribute (rather than `"left"`), so that if the app is used with right-to-left (RTF) languages, the drawer appears on the right rather than the left side.

  ```
  android:layout_gravity="start"
  ```

- Use the `android:fitsSystemWindows="true"` attribute to set the padding of the `DrawerLayout` and the `NavigationView` to ensure the contents don't overlay the system windows. `DrawerLayout` uses `fitsSystemWindows` as a sign that it needs to inset its children (such as the main content view), but still draw the top status bar background in that space. As a result, the navigation drawer appears to be overlapping, but not obscuring, the translucent top status bar. The insets you get from `fitsSystemWindows` will be correct on all platform versions to ensure your content does not overlap with system-provided UI components.

The navigation drawer header

The `NavigationView` specifies the layout for the *header* of the navigation drawer with the attribute `app:headerLayout="@layout/nav_header_main"`. The **nav_header_main.xml** file defines the layout of this header to include an `ImageView` and a `TextView`, which is typical for a navigation drawer, but you could also include other Views.

Tip: The header's height should be 160dp, which you should extract into a dimension resource (`nav_header_height`).

nav_header_main.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:src="@android:drawable/sym_def_app_icon" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:text="@string/my_app_title"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

</LinearLayout>
```

The app bar layout

The `include` tag in the `activity_main` layout includes the `app_bar_main` layout, which uses a CoordinatorLayout as its root. The **app_bar_main.xml** layout file defines the app bar layout with the Toolbar class as shown previously in the chapter about menus. It also defines a floating action button, and uses an `include` tag to include the `content_main` (**content_main.xml**) layout:

app_bar_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.example.android.navigationexperiments.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

Note the following:

- The `app_bar_main` layout uses a CoordinatorLayout as its root, and includes the `content_main` layout.
- The `app_bar_main` layout uses the `android:fitsSystemWindows="true"` attribute to set the padding of the app bar to ensure that it doesn't overlay the system windows such as the status bar.

The content layout for the main activity screen

The above layout uses an `include` tag to include the `content_main` layout, which defines the layout of the main activity screen (**content_main.xml**). In the example layout below, the main activity screen shows a TextView that displays the string "Hello World!":

content_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.example.android.navigationexperiments.MainActivity"
    tools:showIn="@layout/app_bar_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

Note the following:

- The `content_main` layout must be the first child in the `DrawerLayout` because the drawer must be on top of the content. In our layout above, the `content_main` layout is included in the `app_bar_main` layout, which is the first child.
- The `content_main` layout uses a `RelativeLayout` view group set to match the parent view's width and height, because it represents the entire UI when the navigation drawer is hidden.
- The layout *behavior* for the `RelativeLayout` is set to the string resource `@string/appbar_scrolling_view_behavior`, which controls the scrolling behavior of the screen in relation to the app bar at the top. This behavior is defined by the AppBarLayout.ScrollingViewBehavior class. This behavior should be used by Views which can scroll vertically—it supports nested scrolling to automatically scroll any AppBarLayout siblings.

Populating the navigation drawer menu

The `NavigationView` in the `activity_main` layout specifies the menu items for the navigation drawer using the following statement:

```
app:menu="@menu/activity_main_drawer"
```

The menu items are defined in the `activity_main_drawer.xml` file, which is located under **app > res > menu**. The `<group></group>` tag defines a *menu group*—a collection of items that share traits, such as whether they are visible, enabled, or checkable. A group must contains one or more `<item></>` elements and be a child of a `<menu>` element, as shown below. In addition to defining each menu item's title with the `android:title` attribute, the file also defines each menu item's icon with the `android:icon` attribute.

The group is defined with the `android:checkableBehavior` attribute. This attribute lets you put interactive elements within the navigation drawer, such as toggle switches that can be turned on or off, and checkboxes and radio buttons that can be selected. The choices for this attribute are:

- `single`: Only one item from the group can be checked. Use for radio buttons.
- `all`: All items can be checked. Use for checkboxes.
- `none`: No items are checkable.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <group android:checkableBehavior="none">
        <item
            android:id="@+id/nav_camera"
            android:icon="@drawable/ic_menu_camera"
            android:title="@string/import_camera" />
        <item
            android:id="@+id/nav_gallery"
            android:icon="@drawable/ic_menu_gallery"
            android:title="@string/gallery" />
        <item
            android:id="@+id/nav_slideshow"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="@string/slideshow" />
        <item
            android:id="@+id/nav_manage"
            android:icon="@drawable/ic_menu_manage"
            android:title="@string/tools" />
    </group>

    <item android:title="@string/communicate">
        <menu>
            <item
                android:id="@+id/nav_share"
                android:icon="@drawable/ic_menu_share"
                android:title="@string/share" />
            <item
                android:id="@+id/nav_send"
                android:icon="@drawable/ic_menu_send"
                android:title="@string/send" />
        </menu>
    </item>

</menu>
```

Setting up the navigation drawer and item listeners

To use a listener for the navigation drawer's menu items, the activity hosting the navigation drawer must implement the OnNavigationItemSelectedListener interface:

1. Implement `NavigationView.OnNavigationItemSelectedListener` in the class definition:

   ```
   public class MainActivity extends AppCompatActivity
                implements NavigationView.OnNavigationItemSelectedListener {
   ...
   }
   ```

   This interface offers the onNavigationItemSelected() method, which is called when an item in the navigation drawer menu item is tapped. As you enter `OnNavigationItemSelectedListener`, the red warning bulb appears on the left margin.

2. Click the red warning bulb, choose **Implement methods**, and choose the **onNavigationItemSelected(item:MenuItem):boolean** method.

   Android Studio adds a stub for the method:

   ```
   @Override
       public boolean onNavigationItemSelected(MenuItem item) {
           return false;
       }
   }
   ```

   You learn how to use this stub in the next section.

3. Before setting up the navigation item listener, add code to the activity's `onCreate()` method to instantiate the DrawerLayout and NavigationView objects (`drawer` and `navigationView` in the code below):

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   . . .
   DrawerLayout drawer = (DrawerLayout)
                              findViewById(R.id.drawer_layout);
   ActionBarDrawerToggle toggle =
               new ActionBarDrawerToggle(this, drawer, toolbar,
               R.string.navigation_drawer_open,
               R.string.navigation_drawer_close);
   if (drawer != null) {
      drawer.addDrawerListener(toggle);
   }
   toggle.syncState();

   NavigationView navigationView = (NavigationView)
               findViewById(R.id.nav_view);
   if (navigationView != null) {
      navigationView.setNavigationItemSelectedListener(this);
   }
}
```

   The above code instantiates an ActionBarDrawerToggle, which substitutes a special drawable for the activity's **Up** button in the app bar, and links the activity to the DrawerLayout. The special drawable appears as a "hamburger" navigation icon when the drawer is closed, and animates into an arrow as the drawer opens.

   Note: Be sure to use the ActionBarDrawerToggle in support-library-v7.appcompact, not the version in support-library-v4.
   Tip: You can customize the animated toggle by defining the drawerArrowStyle in your ActionBar theme (for more detailed information about the ActionBar theme, see Adding the App Bar in the Android Developer documentation.

   The above code implements addDrawerListener() to listen for drawer open and close events, so that when the user taps custom drawable button, the navigation drawer slides out.

   You must also use the syncState() method of ActionBarDrawerToggle to synchronize the state of the drawer indicator. The synchronization must occur after the DrawerLayout's instance state has been restored, and any other time when the state may have diverged in such a way that the ActionBarDrawerToggle was not notified.

   The above code ends by setting a listener, `setNavigationItemSelectedListener()` , to the navigation drawer to listen for item clicks.

4. The ActionBarDrawerToggle also lets you specify the strings to use to describe the open/close drawer actions for accessibility services. Define the following strings in your strings.xml file:

```
<string name="navigation_drawer_open">Open navigation drawer</string>
<string name="navigation_drawer_close">Close navigation drawer</string>
```

Handling navigation menu item selections

Write code in the onNavigationItemSelected() method stub to handle menu item selections. This method is called when an item in the navigation drawer menu is tapped.

It uses `if` statements to take the appropriate action based on the menu item's `id` , which you can retrieve using the getItemId() method:

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
   DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
   // Handle navigation view item clicks here.
   switch (item.getItemId()) {
      case R.id.nav_camera:
         // Handle the camera import action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_camera));
         return true;
      case R.id.nav_gallery:
         // Handle the gallery action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_gallery));
         return true;
      case R.id.nav_slideshow:
         // Handle the slideshow action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_slideshow));
         return true;
      case R.id.nav_manage:
         // Handle the tools action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_tools));
         return true;
      case R.id.nav_share:
         // Handle the share action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_share));
         return true;
      case R.id.nav_send:
         // Handle the send action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_send));
         return true;
      default:
         return false;
   }
}
```

After the user taps a navigation drawer selection or taps outside the drawer, the DrawerLayout closeDrawer() method closes the drawer.

Lists and carousels

Use a scrolling list, such as a RecyclerView, to provide navigation targets for descendant navigation. Vertically scrolling lists are often used for a screen that lists stories, with each list item acting as a button to each story. For more visual or media-rich content items such as photos or videos, you may want to use a horizontally-scrolling list (also known as a *carousel*). These UI elements are good for presenting items in a collection (for example, a list of news stories).

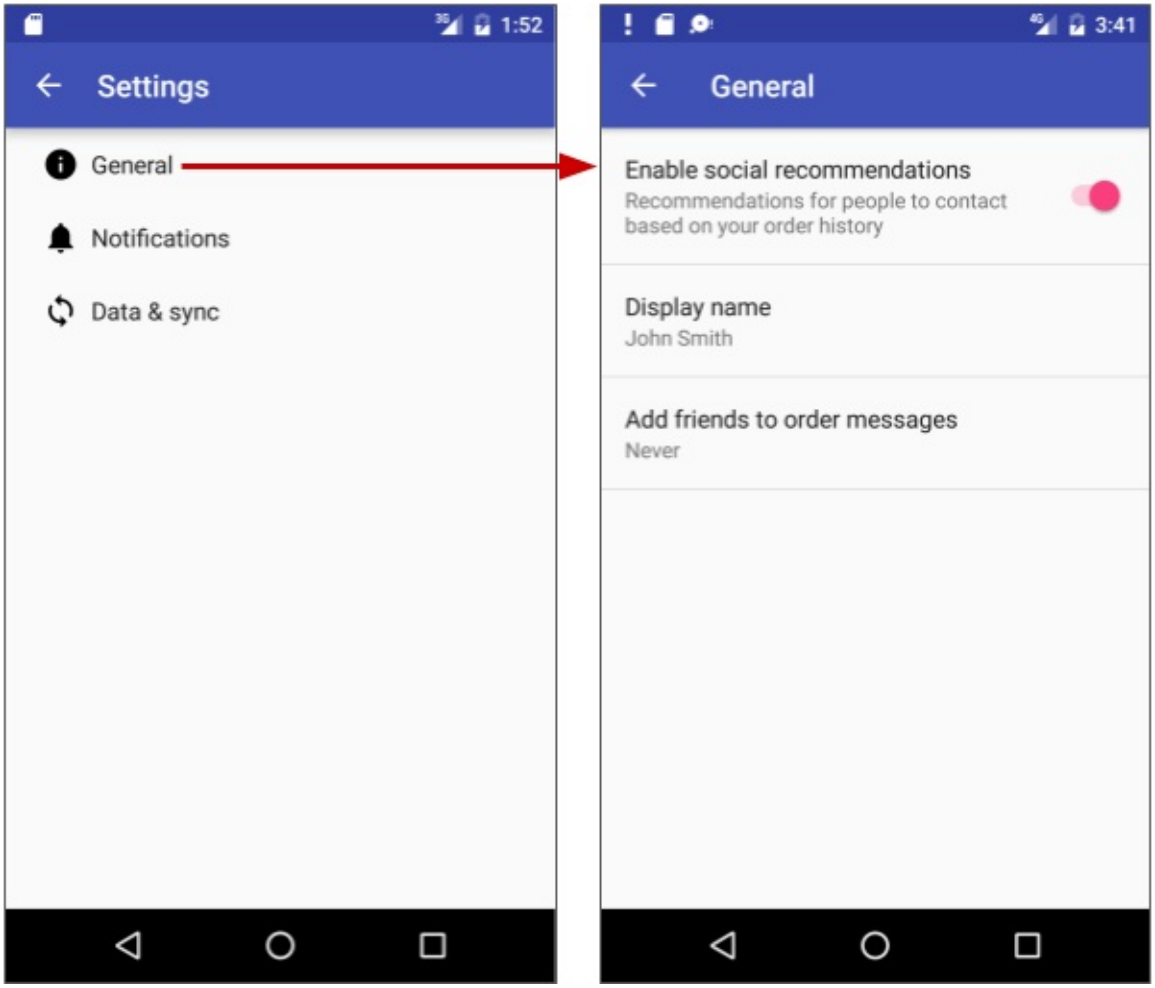
You learn all about RecyclerView in the next chapter.

Master/detail navigation flow

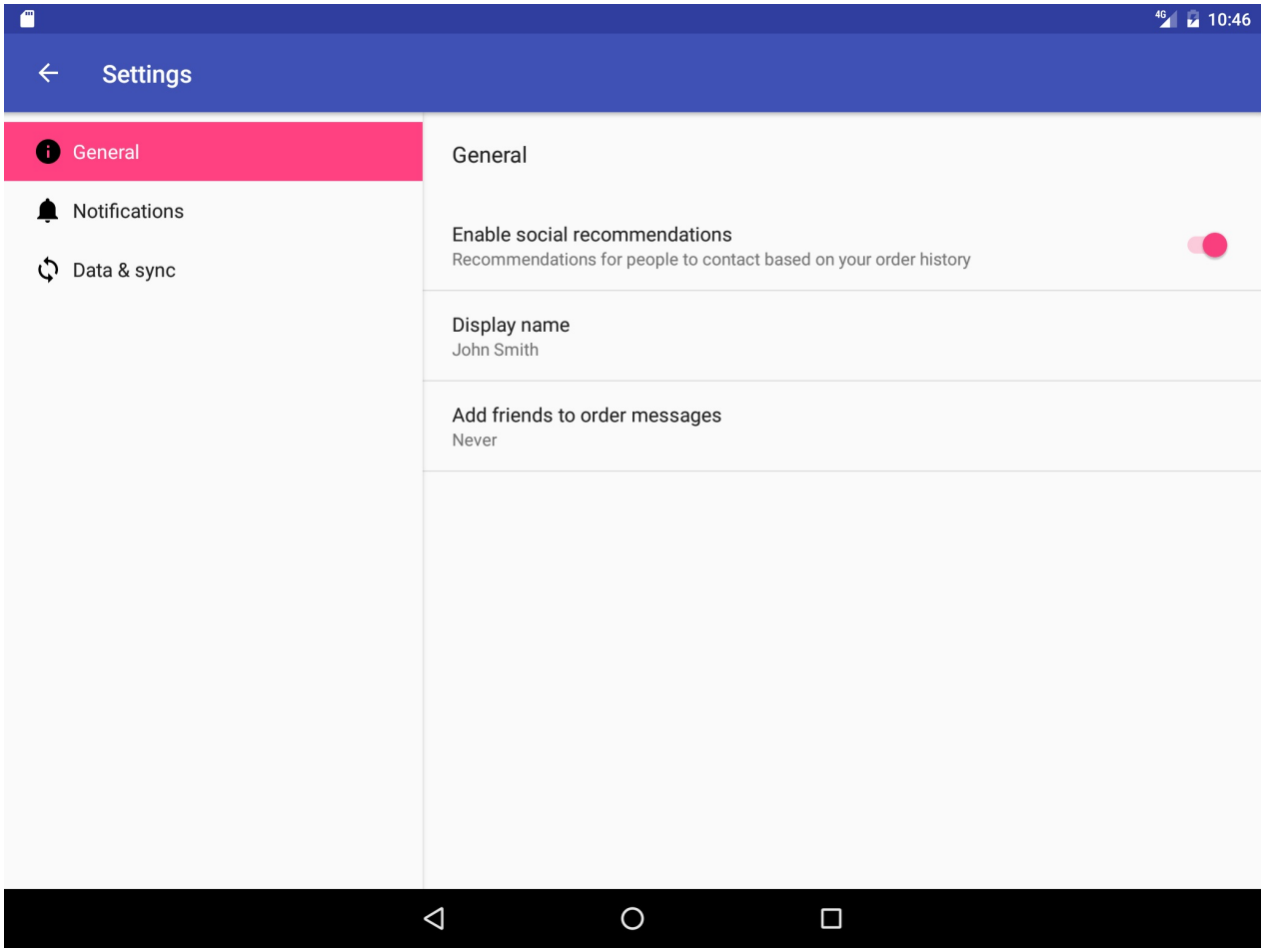
In a master/detail navigation flow, a master screen contains a list of items, and a detail screen shows detailed information about a specific item. Descendant navigation is usually implemented by one of the following:

- Using an Intent that starts an activity representing the detail screen. For more information about Intents, see Intents and Intent Filters in the Android Developer Guide.
- When adding a Settings Activity, you can extend PreferenceActivity to create a two-pane master/detail layout to support large screens, and include fragments within the activity to replace the activity's content with a settings fragment. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as smartphones. You learn about the Settings Activity and PreferenceActivity in a subsequent chapter. For more information about using fragments, see Fragments in the Android Developer Guide.

Smartphones are best suited for displaying one screen at a time—such as a master screen (on the left side of the figure below) and a detail screen (on the right side of the figure below).



On the other hand, tablet displays, especially when viewed in the landscape orientation, are best suited for showing multiple content panes at a time: the master on the left, and the detail to the right, as shown below.

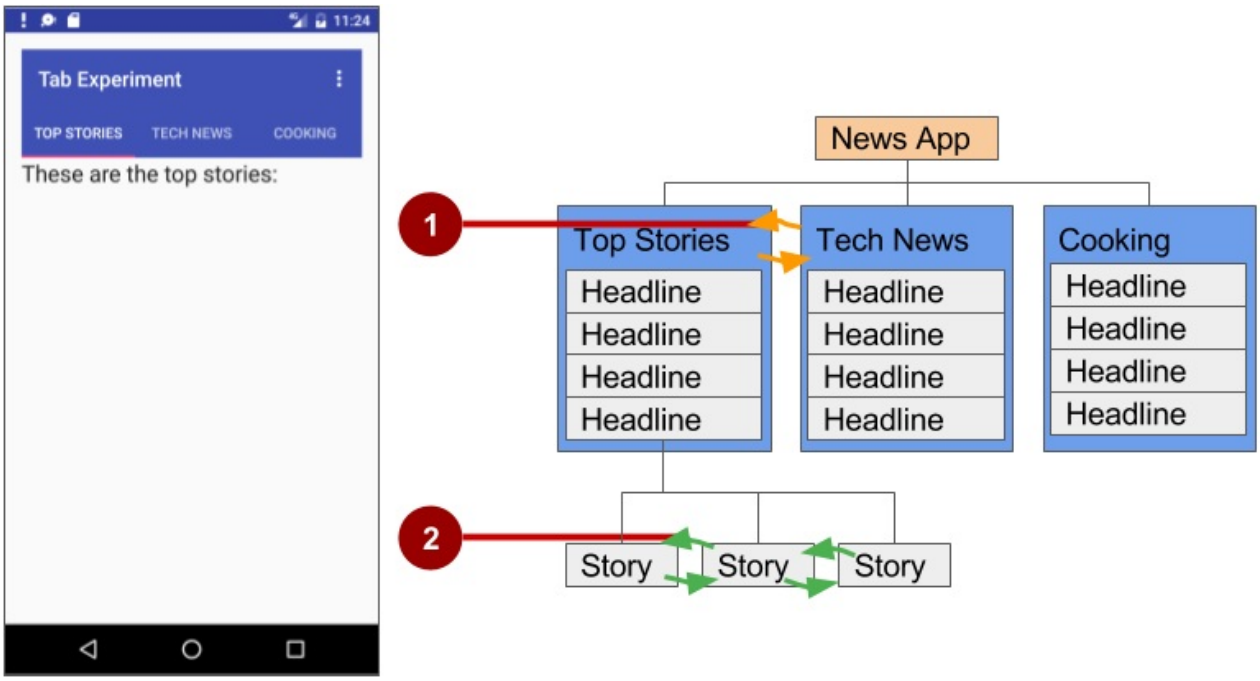


Options menu in the app bar

The app bar typically contains the options menu, which is most often used for navigation patterns for descendant navigation. It may also contain an Up icon for ancestral navigation, a nav icon for opening a navigation drawer, and a filter icon to filter page views. You learned how to set up the options menu and the app bar in a previous chapter.

Lateral navigation with tabs and swipes

With lateral navigation, you enable the user to go from one sibling to another (at the same level in a multitier hierarchy). For example, if your app provides several categories of stories (such as Top Stories, Tech News, and Cooking, as shown in the figure below), you would want to provide your users the ability to navigate from one category to the next, or from one top story to the next, without having to navigate back up to the parent screen.



In the above figure:

1. Lateral navigation from one category screen to another
2. Lateral navigation from one story screen to another

Another example of lateral navigation is the ability to swipe left or right in a Gmail conversation to view a newer or older email in the same Inbox.

You can implement lateral navigation with *tabs* that represent each screen. Tabs appear across the top of a screen, as shown on the left side of the above figure, providing navigation to other screens. Tab navigation is a common solution for lateral navigation from one child screen to another child screen that is a *sibling*—in the same position in the hierarchy and sharing the same parent screen.

Tabs are most appropriate for small sets (four or fewer) of sibling screens. You can combine tabs with swipe views, so that the user can swipe across from one screen to another as well as tap a tab.

Tabs offer two benefits:

- Since there is a single, initially-selected tab, users already have access to that tab's content from the parent screen without any further navigation.
- Users can navigate quickly between related screens, without needing to first revisit the parent.

Keep in mind the following best practices when using tabs:

- Tabs are usually laid out horizontally.
- Tabs should always run along the top of the screen, and should not be aligned to the bottom of the screen.
- Tabs should be persistent across related screens. Only the designated content region should change when tapping a tab, and tab indicators should remain available at all times.
- Switching to another tab should not be treated as history. For example, if a user switches from tab A to tab B, pressing the **Up** button in the app bar should not reselect tab A but should instead return the user to the parent screen.

The key steps for implementing tabs are:

1. Defining the tab layout. The main class used for displaying tabs is TabLayout. It provides a horizontal layout to display tabs. You can show the tabs below the app bar.
2. Implementing a Fragment for each tab content screen. A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own own lifecycle. One benefit of using fragments for the tab content is that you can isolate the code for managing the tab content in the fragment. To learn about fragments, see Fragments in the API Guide.

3. Adding a pager adapter. Use the PagerAdapter class to populate "pages" (screens) inside of a ViewPager, which is a layout manager that lets the user flip left and right through screens of data. You supply an implementation of a PagerAdapter to generate the screens that the view shows. ViewPager is most often used in conjunction with Fragment, which is a convenient way to supply and manage the lifecycle of each screen.
4. Creating an instance of the tab layout, and set the text for each tab.
5. Using `PagerAdapter` to manage screen ("page") views. Each screen is represented by its own fragment.
6. Setting a listener to determine which tab is tapped.

There are standard adapters for using fragments with the ViewPager:

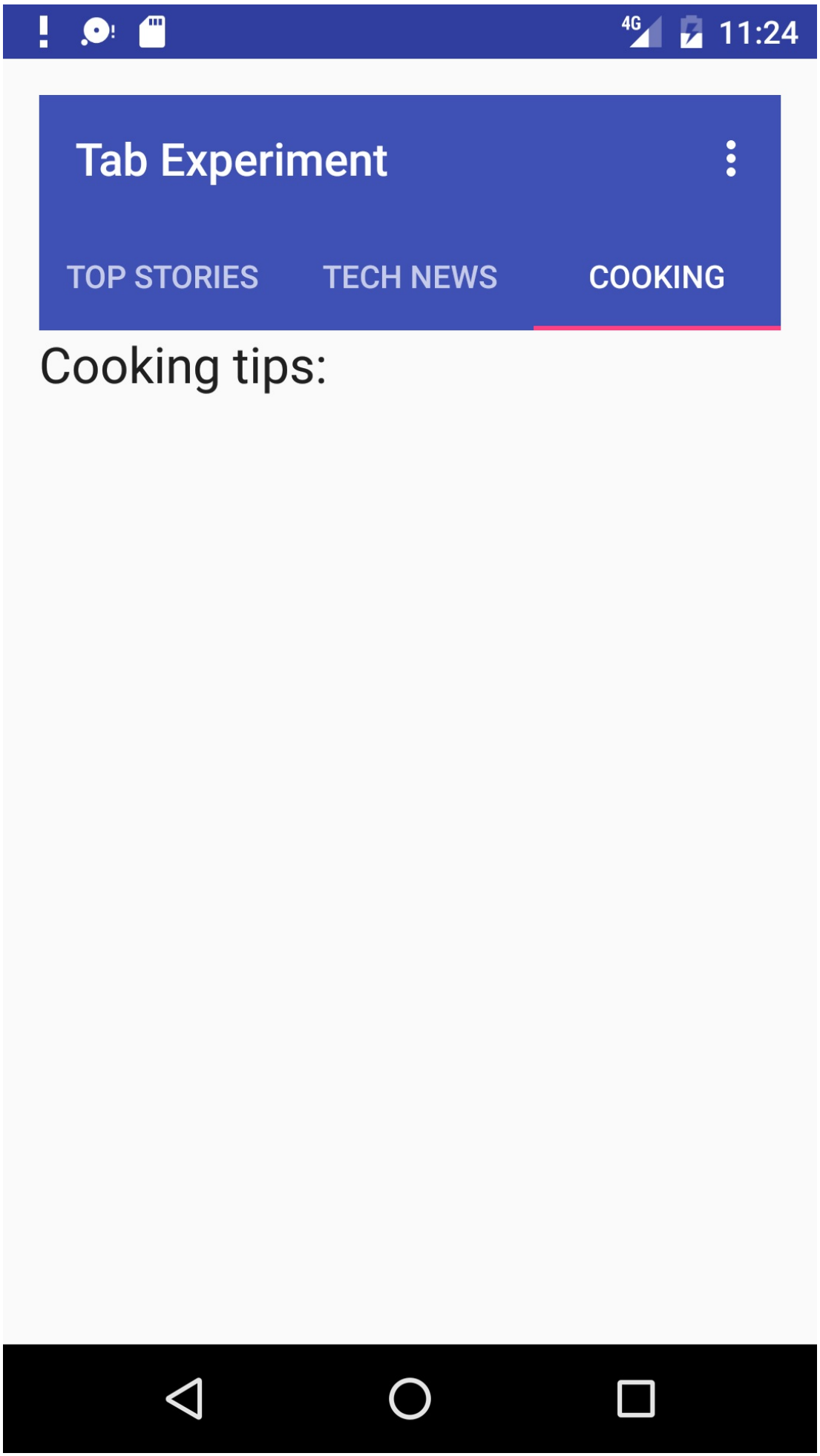
- FragmentPagerAdapter: Designed for navigating between sibling screens (pages) representing a fixed, small number of screens.
- FragmentStatePagerAdapter: Designed for paging across a collection of screens (pages) for which the number of screens is undetermined. It destroys fragments as the user navigates to other screens, minimizing memory usage. The app for this practical challenge uses FragmentStatePagerAdapter.

Defining tab layout

To use a TabLayout, you can design the main activity's layout to use a `Toolbar` for the app bar, a `TabLayout` for the tabs below the app bar, and a `ViewPager` within the root layout to switch child views. The layout should look similar to the following, assuming each child view fills the screen:

```
<android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

    <android.support.design.widget.TabLayout
        android:id="@+id/tab_layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/toolbar"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>

    <android.support.v4.view.ViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:layout_below="@id/tab_layout"/>
```

For each child view, create a layout file such as tab_fragment1.xml, tab_fragment2.xml, tab_fragment3.xml, and so on.


11:24
Tab Experiment
TOP STORIES
TECH NEWS
COOKING
Cooking tips:

Implementing each fragment

A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own own lifecycle. To learn about fragments, see Fragments in the API Guide.

Add a class for each fragment (such as TabFragment1.java, TabFragment2.java, and TabFragment3.java) representing a screen the user can visit by clicking a tab. Each class should extend Fragment and inflate the layout associated with the screen (`tab_fragment1` , `tab_fragment2` , and `tab_fragment3`). For example, TabFragment1.java looks like this:

```
public class TabFragment1 extends Fragment {

   @Override
   public View onCreateView(LayoutInflater inflater,
                   ViewGroup container, Bundle savedInstanceState) {
      return inflater.inflate(R.layout.tab_fragment1, container, false);
   }
}
```

Adding a pager adapter

Add a `PagerAdapter` that extends FragmentStatePagerAdapter and:

1. Defines the number of tabs.
2. Uses the getItem() method of the Adapter class to determine which tab is clicked.
3. Uses a `switch case` block to return the screen (page) to show based on which tab is clicked.

```
public class PagerAdapter extends FragmentStatePagerAdapter {
    int mNumOfTabs;

    public PagerAdapter(FragmentManager fm, int NumOfTabs) {
        super(fm);
        this.mNumOfTabs = NumOfTabs;
    }

    @Override
    public Fragment getItem(int position) {

        switch (position) {
            case 0:
                return new TabFragment1();
            case 1:
                return new TabFragment2();
            case 2:
                return new TabFragment3();
            default:
                return null;
        }
    }

    @Override
    public int getCount() {
        return mNumOfTabs;
    }
}
```

Creating an instance of the tab layout

In the `onCreate()` method of the main activity, create an instance of the tab layout from the `tab_layout` element in the layout, and set the text for each tab using addTab():

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   ...
   // Create an instance of the tab layout from the view.
   TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);
   // Set the text for each tab.
   tabLayout.addTab(tabLayout.newTab().setText("Top Stories"));
   tabLayout.addTab(tabLayout.newTab().setText("Tech News"));
   tabLayout.addTab(tabLayout.newTab().setText("Cooking"));
   // Set the tabs to fill the entire layout.
   tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);
   // Use PagerAdapter to manage page views in fragments.
   ...
}
```

Extract string resources for the tab text set by `setText()`:

- `"Top Stories"` to `tab_label1`
- `"Tech News"` to `tab_label2`
- `"Cooking"` to `tab_label3`

Managing screen views in fragments and set a listener

Use `PagerAdapter` in the main activity's `onCreate()` method to manage screen ("page") views in the fragments. Each screen is represented by its own fragment. You also need to set a listener to determine which tab is tapped. The following code should appear after the code from the previous section in the `onCreate()` method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   ...
   // Use PagerAdapter to manage page views in fragments.
   // Each page is represented by its own fragment.
   // This is another example of the adapter pattern.
   final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
   final PagerAdapter adapter = new PagerAdapter
               (getSupportFragmentManager(), tabLayout.getTabCount());
   viewPager.setAdapter(adapter);
   // Setting a listener for clicks.
   viewPager.addOnPageChangeListener(new
               TabLayout.TabLayoutOnPageChangeListener(tabLayout));
   tabLayout.addOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
      @Override
      public void onTabSelected(TabLayout.Tab tab) {
         viewPager.setCurrentItem(tab.getPosition());
      }

      @Override
         public void onTabUnselected(TabLayout.Tab tab) {

      }

      @Override
         public void onTabReselected(TabLayout.Tab tab) {
      }
   });
}
```

Using ViewPager for swipe views (horizontal paging)

The ViewPager is a layout manager that lets the user flip left and right through "pages" (screens) of content. ViewPager is most often used in conjunction with Fragment, which is a convenient way to supply and manage the lifecycle of each "page". ViewPager also provides the ability to swipe "pages" horizontally.

In the previous example, you used a `ViewPager` within the root layout to switch child screens. This provides the ability for the user to swipe from one child screen to another. Users are able to navigate to sibling screens by touching and dragging the screen horizontally in the direction of the desired adjacent screen.

Swipe views are most appropriate where there is some similarity in content type among sibling pages, and when the number of siblings is relatively small. In these cases, this pattern can be used along with tabs above the content region to indicate the current page and available pages, to aid discoverability and provide more context to the user.

Tip: It's best to avoid horizontal paging when child screens contain horizontal panning surfaces (such as maps), as these conflicting interactions may deter your screen's usability.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Using the App Bar and Tabs for Navigation

Learn more

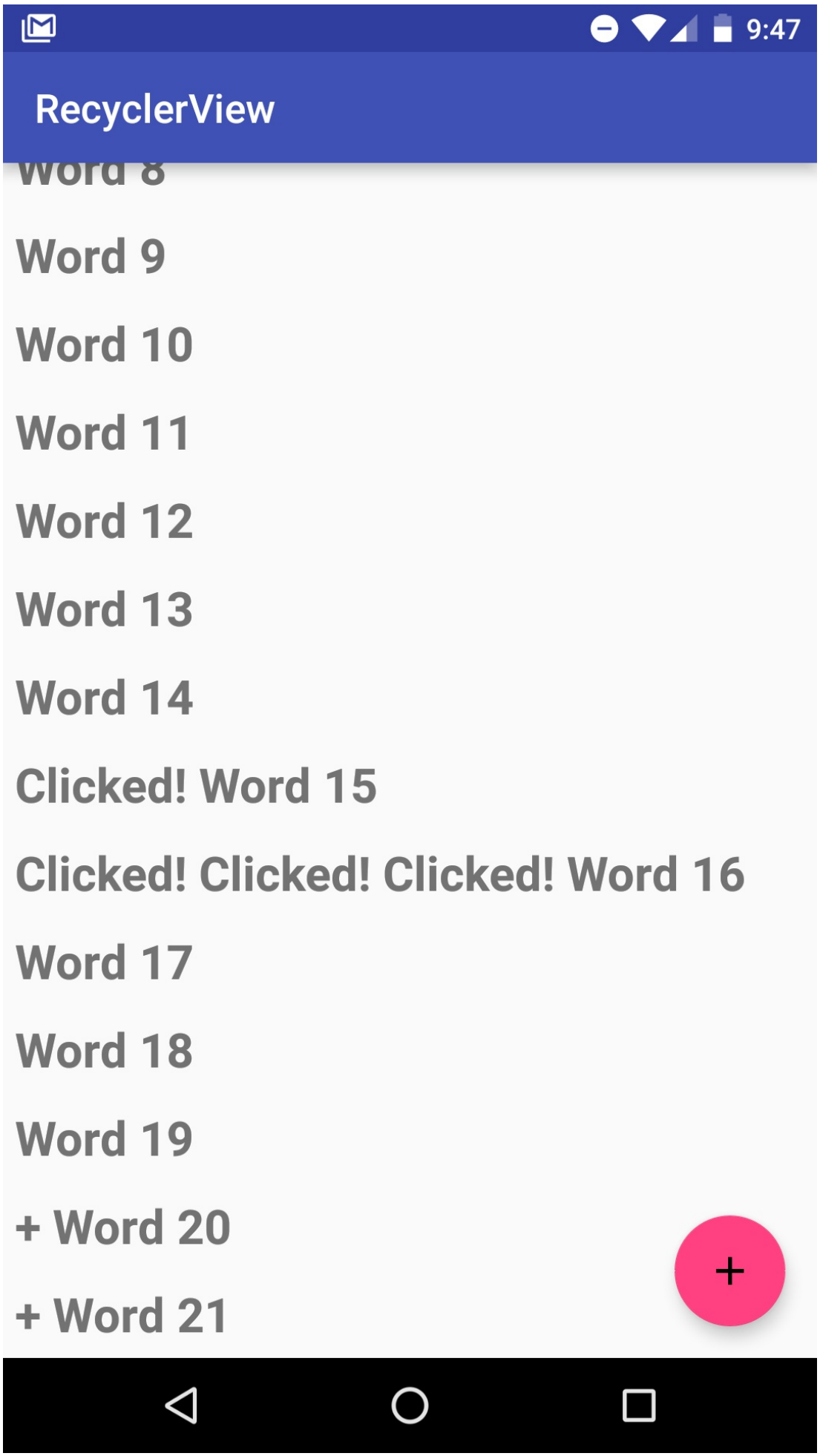
- Best Practices for Interaction and Engagement
  - Designing Effective Navigation
  - Implementing Effective Navigation
  - Creating Swipe Views with Tabs
  - Creating a Navigation Drawer
  - Providing Up Navigation
  - Implementing Descendant Navigation
- Best Practices for User Interface
- Design - Patterns - Navigation
- Navigation with Back and Up
- Action Bar
- Adding the Action (App) Bar tutorial
- Menus

4.4: RecyclerView

Contents:

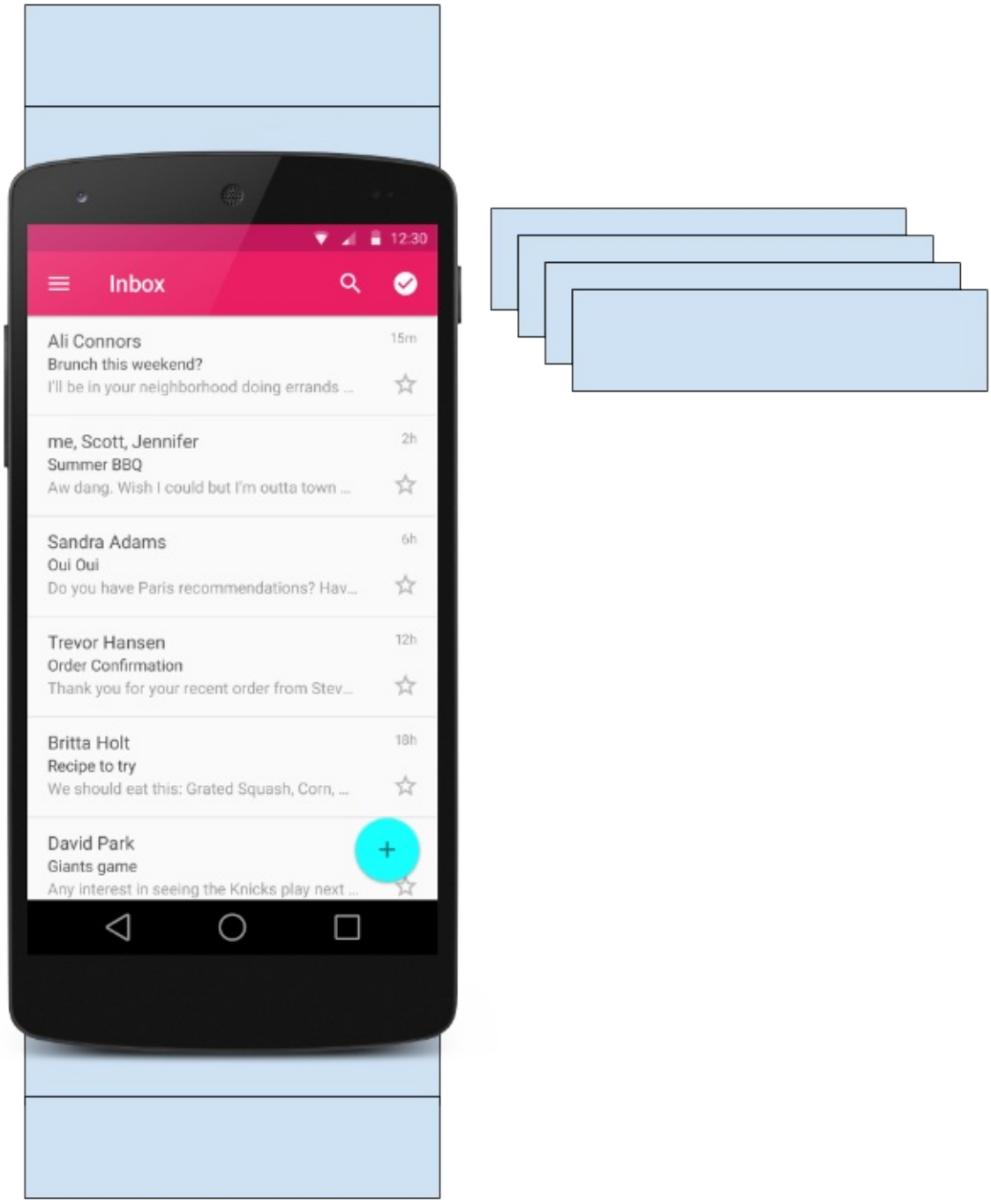
- RecyclerView components
- Data
- RecyclerView
- Item Layout
- Layout Manager
- Animations
- Adapter
- ViewHolder
- Implementing a RecyclerView
- 1. Add the dependency to app/build.gradle
- 2. Add a RecyclerView to your activity's layout
- 3. Create the layout for one item
- 4. Create an adapter with a view holder
- 5. Implement the view holder class
- 6. Create the RecyclerVew
- Related practical
- Learn more

When you display a large number of items in a scrollable list, most items are not visible. For example, in a long list of words or many news headlines, the user only sees a small number of list items at a time.


9:47
RecyclerView
Word 8
Word 9
Word 10
Word 11
Word 12
Word 13
Word 14
Clicked! Word 15
Clicked! Clicked! Clicked! Word 16
Word 17
Word 18
Word 19
+ Word 20
+ Word 21

Or you may have a dataset that changes as the user interacts with it. If you create a new view every time the data changes, that's also a lot of views, even for a small dataset.

From a performance perspective, you want to minimize the number of views kept around at any given point (Memory), and the number of views you have to create (Time). Both of these goals can be accomplished by creating somewhat more views than the user can see on the screen, and cache and reuse previously created views with different data as they scroll in and out of view.



The RecyclerView class is a more advanced and flexible version of ListView. It is a container for displaying large data sets that can be scrolled very efficiently by maintaining a limited number of views.

Use the RecyclerView widget when you need to display a large amount of scrollable data, or data collections whose elements change at runtime based on user action or network events.

RecyclerView components

To display your data in a RecyclerView, you need the following parts:

- **Data.** It doesn't matter where the data comes from. You can create the data locally, as you do in the practical, get it from a database on the device as you will do in a later practical, or pull it from the cloud.
- **A RecyclerView.** The scrolling list that contains the list items.

  An instance of RecyclerView as defined in your activity's layout file to act as the container for the views.

- **Layout for one item of data.** All list items look the same, so you can use the same layout for all of them. The item layout has to be created separately from the activity's layout, so that one item view at a time can be created and filled with data.
- **A layout manager.** The layout manager handles the organization (layout) of user interface components in a view. All view groups have layout managers. For the LinearLayout, the Android system handles the layout for you. RecyclerView requires an explicit layout manager to manage the arrangement of list items contained within it. This layout could be vertical, horizontal, or a grid.

  The layout manager is an instance of Recyclerview.LayoutManager to organize the layout of the items in the RecyclerView

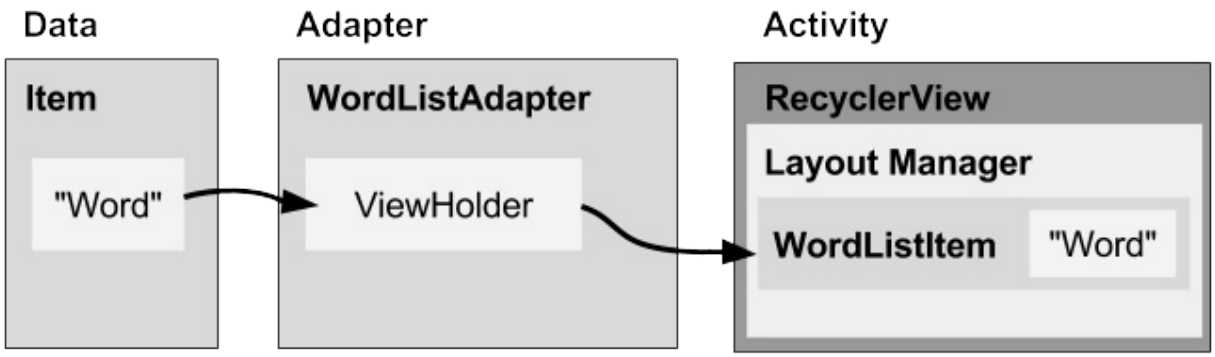
- **An adapter.** The adapter connects your data to the RecyclerView. It prepares the data and how will be displayed in a view holder. When the data changes, the adapter updates the contents of the respective list item view in the RecyclerView.

  And an adapter is an extension of RecyclerView.Adapter. The adapter uses a ViewHolder to hold the views that constitute each item in the RecyclerView, and to bind the data to be displayed into the views that display it.

- **A view holder.** The view holder extends the ViewHolder class. It contains the view information for displaying one item from the item's layout.

  A view holder used by the adapter to supply data, which is an extension of RecyclerView.ViewHolder

The diagram below shows the relationship between these compoments.



Data

Any displayable data can be shown in a RecyclerView.

- Text
- Images
- Icons

Data can come from any source.

- Created by the app. For example, scrambled words for a game.
- From a local database. For example, a list of contacts.
- From cloud storage or the internet. For example news headlines.

RecyclerView

A RecyclerView is:

- A View group for a scrollable container
- Ideal for long lists of similar items
- Uses only a limited number of views that are re-used when they go off-screen. This saves memory and makes it faster to update list items as the user scrolls through data, because it is not necessary to create a new view for every item that appears.
- In general, the RecyclerView keeps as many item views as fit on the screen, plus a few extra at each end of the list to make sure that scrolling is fast and smooth.

Item Layout

The layout for a list item is kept in a separate file so that the adapter can create item views and edit their contents independently from the layout of the activity.

Layout Manager

A layout manager positions item views inside a view group, such as the RecyclerView and determines when to reuse item views that are no longer visible to the user. To reuse (or recycle) a view, a layout manager may ask the adapter to replace the contents of the view with a different element from the dataset. Recycling views in this manner improves performance by avoiding the creation of unnecessary views or performing expensive findViewById() lookups.

RecyclerView provides these built-in layout managers:

- LinearLayoutManager shows items in a vertical or horizontal scrolling list.
- GridLayoutManager shows items in a grid.
- StaggeredGridLayoutManager shows items in a staggered grid.

To create a custom layout manager, extend the RecyclerView.LayoutManager class.

Animations

Animations for adding and removing items are enabled by default in RecyclerView. To customize these animations, extend the RecyclerView.ItemAnimator class and use the RecyclerView.setItemAnimator() method.

Adapter

An Adapter helps two incompatible interfaces to work together. In the RecyclerView, the adapter connects data with views. It acts as an intermediary between the data and the view. The Adapter receives or retrieves the data, does any work required to make it displayable in a view, and places the data in a view.

For example, the adapter may receive data from a database as a Cursor object, extract the the word and its definition, convert them to strings, and place the strings in an item view that has two text views, one for the word and one for the definition. You will learn more about cursors in a later chapter.

The RecyclerView.Adapter implements a view holder, and must override the following callbacks:

- `onCreateViewHolder()` inflates an item view and returns a new view holder that contains it. This method is called when the RecyclerView needs a new view holder to represent and item.
- `onBindViewHolder()` sets the contents of an item at a given position in the RecyclerView. This is called by the RecyclerView, for example, when a new item scrolls into view.
- `getItemCount()` returns the total number of items in the data set held by the adapter.

View holder

A RecyclerView.ViewHolder describes an item view and metadata about its place within the RecyclerView. Each view holder holds one set of data. The adapter adds data to view holders for the layout manager to display.

You define your view holder layout in an XML resource file. It can contain (almost) any type of view, including clickable elements.

Implementing a Recycler View

Implementing a RecyclerView requires the following steps:

1. Add the RecyclerView dependency to the app's app/build.gradle file.
2. Add the RecyclerView to the activity's layout
3. Create a layout XML file for one item
4. Extend RecyclerView.Adapter and implement onCrateViewHolder and onBindViewHolder methods.
5. Extend RecyclerView.ViewHolder to create a view holder for your item layout. You can add click behavior by overriding the onClick method.
6. In your activity, In the onCreate method, create a RecyclerView and initialize it with the adapter and a layout manager.

1. Add the dependency to app/build.gradle

Add the recycler view library to your app/build.gradle file as a dependency. Look at the chapter on support libraries or the RecyclerView practical, if you need detailed instructions.

```
dependencies {
    ...
    compile 'com.android.support:recyclerview-v7:24.1.1'
    ...
}
```

2. Add a RecyclerView to your activity's layout

Add the RecyclerView in your activity's layout file.

```
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    </android.support.v7.widget.RecyclerView>
```

Use the recycler view from the support library to be compatible with older devices. The only required attributes are the id, along with the width and height. Customize the items, not this view group.

3. Create the layout for one item

Create an XML resource file and specify the layout of one item. This will be used by the adapter to create the view holder.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="6dp">

    <TextView
        android:id="@+id/word"
        style="@style/word_title" />

</LinearLayout>
```

The text view has a `@style` element. A style is a collection of properties that specifies the look of a view. You can use styles to share display attributes with multiple views. An easy way to create a style is to extract the style of a UI element that you already created. For example, after styling a TextView, **Right-click > Refactor > Extract > Style** on the element and follow the dialog prompts. More details on styles are in the practical and in a later chapter.

4. Create an adapter with a view holder

Extend RecyclerView.Adapter and implement the onCrateViewHolder and onBindViewHolder methods.

Create a new Java class with the following signature:

```
public class WordListAdapter extends RecyclerView.Adapter<WordListAdapter.WordViewHolder> {}
```

In the constructor, get an inflater from the current contex, and your data.

```
    public WordListAdapter(Context context, LinkedList<String> wordList) {
        mInflater = LayoutInflater.from(context);
        this.mWordList = wordList;
    }
```

For this adapter, you have to implement 3 methods.

- onCreateViewHolder() creates a view and returns it.

  ```
  @Override
  public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType){
      // Inflate an item view.
      View mItemView = mInflater.inflate(R.layout.wordlist_item, parent, false);
      return new WordViewHolder(mItemView, this);
  }
  ```

- onBindViewHolder() associates the data with the view holder for a given position in the RecyclerView.

  ```
  @Override
  public void onBindViewHolder(WordViewHolder holder, int position) {
      // Retrieve the data for that position
      String mCurrent = mWordList.get(position);
      // Add the data to the view
      holder.wordItemView.setText(mCurrent);
  }
  ```

- getItemCount() returns to number of data items available for displaying.

  ```
   @Override
  public int getItemCount() {
      return mWordList.size();
  }
  ```

5. Implement the view holder class

Extend RecyclerView.ViewHolder to create a view holder for your item layout. You can add click behavior by overriding the onClick method.

This class is usually defined as an inner class to the adapter and extends RecyclerView.ViewHolder.

```
class WordViewHolder extends RecyclerView.ViewHolder {}
```

If you want to add click handling, you need to implement a click listener. One way to do this is to have the view holder implement the click listener methods.

```
// Extend the signature of WordVewHolder to implement a click listener.
class WordViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {}
```

In its constructor, the view holder has to inflate its layout, associate with its adapter, and, if applicable, set a click listener.

```
public WordViewHolder(View itemView, WordListAdapter adapter) {
    super(itemView);
    wordItemView = (TextView) itemView.findViewById(R.id.word);
    this.mAdapter = adapter;
    itemView.setOnClickListener(this);
}
```

And, if you implementing onClickListener, you also have to implement onClick().

```
@Override
public void onClick(View v) {
    wordItemView.setText ("Clicked! "+ wordItemView.getText());
}
```

Note that to attach click listeners to other elements of the view holder, you do that dynamically in onBindViewHolder. (You will do this a later practical, when you will be extending the recycler view code from the practical.)

6. Create the RecyclerView

Finally, to tie it all together, in your activity's onCreate() method:

1. Get a handle to the RecyclerView.
   ```
   mRecyclerView = (RecyclerView) findViewById(R.id.recyclerview);
   ```
2. Create an adapter and supply the data to be displayed.
   ```
   mAdapter = new WordListAdapter(this, mWordList);
   ```
3. Connect the adapter with the recycler view.
   ```
   mRecyclerView.setAdapter(mAdapter);
   ```
4. Give the recycler view a default layout manager.
   ```
   mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
   ```

RecyclerView is an efficient way for displaying scrolling list data. It uses the adapter pattern to connect data with list item views. To implement a RecyclerView you need to create an adapter and a view holder, and the methods that take the data and add it to the list items.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Create a RecyclerView

Learn more

- RecyclerView
- RecyclerView class
- RecyclerView.Adapter class
- RecyclerView.ViewHolder class
- RecyclerView.LayoutManager class

5.1: Drawables, Styles, and Themes

Table of Contents:

- Introduction
- Drawables
- Images
- Styles
- Themes
- Related practical
- Learn more

In this chapter you learn how to use *drawables*, which are compiled images that you can use in your app. Android provides classes and resources to help you include rich images in your application with a minimal impact to your app's performance.

You also learn how to use styles and themes to provide a consistent appearance to all the elements in your app while reducing the amount of code.

Drawables

A *drawable* is a graphic that can be drawn to the screen. You retrieve drawables using APIs such as `getDrawable(int)`, and you apply a drawable to an XML resource using attributes such as `android:drawable` and `android:icon`.

Android includes several types of drawables, most of which are covered in this chapter.

Covered in this chapter:	Not covered in this chapter:
• Image files • Nine-patch files • Layer lists • Shape drawables • State lists • Level lists • Transition drawables • Vector drawables	• Scale drawables • Inset drawables • Clip drawables

Using drawables

To display a drawable, use the `ImageView` class to create a View. In the `<ImageView>` element in your XML file, define how the drawable is displayed and where the drawable file is located. For example, this `ImageView` displays an image called "birthdaycake.png":

```
<ImageView
      android:id="@+id/tiles"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/birthdaycake" />
```

About the `<ImageView>` attributes:

- The `android:id` attribute sets a shortcut name that you use to call the image later.
- The `android:layout_width` and `android:layout_height` attributes specify the size of the View. In this example the height and width are set to `wrap_content`, which means the View is only big enough to enclose the image within it, plus padding.
- The `android:src` attribute gives the location where this image is stored. If you have versions of the image that are

appropriate for different screen resolutions, store them in folders named res/drawable-[density]/. For example, store a version of birthdaycake.png appropriate for hdpi screens in res/drawable-hdpi/birthdaycake.png. For more information, see the multiple-screens guide.

- `<ImageView>` also has attributes that you can use to crop your image if it is too large or has a different aspect ratio than the layout or the View. For complete details, see the `ImageView` class documentation.

To represent a drawable in your app, use the `Drawable` class or one of its subclasses. For example, this code retrieves the birthdaycake.png image as a `Drawable`:

```
Resources res = getResources();
Drawable drawable = res.getDrawable(R.drawable.birthdaycake);
```

Image files

An *image file* is a generic bitmap file. Android supports image files in several formats: WebP (preferred), PNG (preferred), and JPG (acceptable). GIF and BMP formats are supported, but discouraged.

The WebP format is fully supported from Android 4.2. WebP compresses better than other formats for lossless and lossy compression, potentially resulting in images more than 25% smaller than JPEG formats. You can convert existing PNG and JPEG images into WebP format before upload. For more about WebP, see the WebP documentation.

Store image files in the res/drawable folder. Use them with the `android:src` attribute for an `ImageView` and its descendants, or to create a BitmapDrawable class in Java code.

Be aware that images look different on screens with different pixel densities and aspect ratios. For information on supporting different screen sizes, see Speeding up your app, below, and the screen sizes guide.

Note: Always use appropriately sized images, because images can use up a lot of disk space and affect your app's performance.

Nine-patch files

A *9-patch* is a PNG image in which you define stretchable regions. Use a 9-patch as the background image for a View to make sure the View looks correct for different screen sizes and orientations.

For example, in a View that has `layout_width` set to `"wrap_content"`, the View stays big enough to enclose its content (plus padding). If you use a normal PNG image as the background image for the View, the image might be too small for the for the View on some devices, because the View stretches to accommodate the content inside it. If you use a 9-patch image instead, the 9-patch stretches as the View stretches.

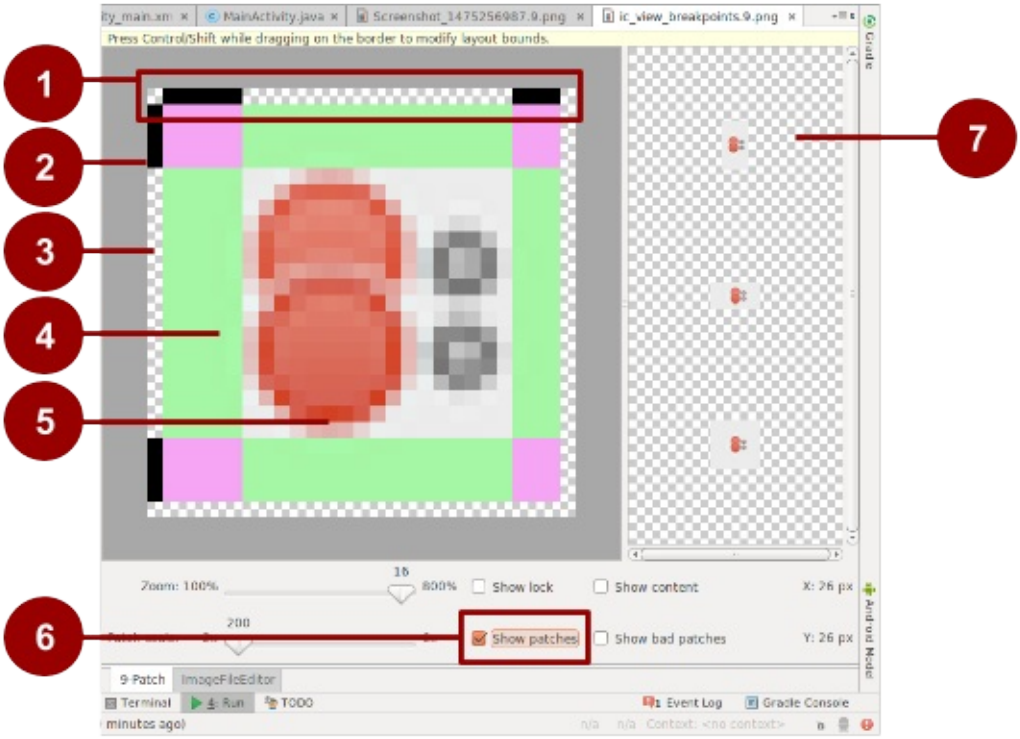
Android's standard Button widget is an example of a View that uses a 9-patch as its background image. The 9-patch stretches to accommodate the text or image inside the button.

Save 9-patch files with a .9.png extension and store them in the res/drawable folder. Use them with the `android:src` attribute for an `ImageView` and its descendants, or to create a `NinePatchDrawable` class in Java code.

To create a 9-patch, use the Draw 9-Patch tool in Android Studio. The tool lets you start with a regular PNG and define a 1-pixel border around the image in places where it's okay for the Android system to stretch the image if needed. To use the tool:

1. Put a PNG file into the res/drawable folder. (To do this, copy the image file into the app/src/main/res/drawable folder of your project.)
2. In Android Studio, right-click the file and choose **Create 9-Patch file**. Android Studio saves the file with a .9.png extension.
3. In Android Studio, double-click the .9.png file to open the editor.
4. Specify which regions of the image are okay to stretch.

# #



1. Border to indicate which regions are okay to stretch for width (horizontally).

   For example, in a View that is wider than the image, the green stripes on the left- and right-hand sides of this 9-patch can be stretched to fill the View. Places that can stretch are marked with black. Click to turn pixels black.

2. Border to indicate regions that are okay to stretch for height (vertically). For example, in a View that is taller then the image, the green stripes on the top and bottom of this 9-patch can be stretched to fill the View.
3. Turn off pixels by shift-clicking (ctrl-click on Mac).
4. Stretchable area.
5. Not stretchable.
6. Check **Show patches** to preview the stretchable patches in the drawing area.
7. Previews of stretched image.
   Tip: Make sure that stretchable regions are at least 2x2 pixels in size. Otherwise, they may disappear when the image is scaled down.

For a more detailed discussion about how to create a 9-patch file with stretchable regions, see the 9-patch guide.

Layer list drawables

In Android you can build up an image by layering other images together, just as you can in Gimp and other image-manipulation programs. Each layer is represented by an individual drawable. The drawables that make up a single image are organized and managed in a `<layer-list>` element in XML. Within the `<layer-list>`, each drawable is represented by an `<item>` element.

Layers are drawn on top of each other in the order defined in the XML file, which means that the last drawable in the list is drawn on top. For example, this layer list drawable is made up of three drawables superimposed on each other:

In the following XML, which defines this layer list, the `android_blue` image is defined last, so it's drawn last and shown on top:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
      <bitmap android:src="@drawable/android_red"
        android:gravity="center" />
    </item>
    <item android:top="10dp" android:left="10dp">
      <bitmap android:src="@drawable/android_green"
        android:gravity="center" />
    </item>
    <item android:top="20dp" android:left="20dp">
      <bitmap android:src="@drawable/android_blue"
        android:gravity="center" />
    </item>
</layer-list>
```

A `LayerDrawable` is a drawable object that manages an array of other drawables. For more information about how to use a layer list drawable, see the layer list guide.

Shape drawables

A shape drawable is a rectangle, oval, line, or ring that you define in XML. You specify the size and style of the shape using XML attributes.

For example, this XML file creates a rectangle with rounded corners and a color gradient. The rectangle's fill color shifts from white (`#000000`) in the lower left corner to blue (`#0000dd)` in the upper right corner. The `angle` attribute determines how the gradient is tilted:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <corners android:radius="8dp" />
    <gradient
        android:startColor="#000000"
        android:endColor="#0000dd"
        android:angle="45"/>
    <padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
        android:bottom="7dp" />
</shape>
```

Assuming that the shape drawable XML file is saved at res/drawable/gradient_box.xml, the following layout XML applies the shape drawable as the background to a View:

```
<TextView
    android:background="@drawable/gradient_box"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content" />
```

The following code shows how to programmatically get the shape drawable and use it as the background for a View, as an alternative to defining the background attribute in XML:

```
Resources res = getResources();
Drawable shape = res. getDrawable(R.drawable.gradient_box);

TextView tv = (TextView)findViewByID(R.id.textview);
tv.setBackground(shape);
```

You can set other attributes for a shape drawable. The complete syntax is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape=["rectangle" | "oval" | "line" | "ring"] >
    <!-- If it's a line, the stroke element is required. -->
    <corners
        android:radius="integer"
        android:topLeftRadius="integer"
        android:topRightRadius="integer"
        android:bottomLeftRadius="integer"
        android:bottomRightRadius="integer" />
    <gradient
        android:angle="integer"
        <!-- The angle must be 0 or a multiple of 45 -->
        android:centerX="float"
        android:centerY="float"
        android:centerColor="integer"
        android:endColor="color"
        android:gradientRadius="integer"
        android:startColor="color"
        android:type=["linear" | "radial" | "sweep"]
        android:useLevel=["true" | "false"] />
    <padding
        android:left="integer"
        android:top="integer"
        android:right="integer"
        android:bottom="integer" />
    <size
        android:width="integer"
        android:height="integer" />
    <solid
        android:color="color" />
    <stroke
        android:width="integer"
        android:color="color"
        android:dashWidth="integer"
        android:dashGap="integer" />
</shape>
```

For details about these attributes, see the shape drawable reference.

State list drawables

A `StateListDrawable` is a drawable object that uses a different image to represent the same object, depending on what state the object is in. For example, a Button widget can exist in one of several states (pressed, focused on, hovered over, or none of these). Using a state list drawable, you can provide a different background image for each state.

You describe the state list in an XML file. Each graphic is represented by an `<item>` element inside a single `<selector>` element. Each `<item>` uses a `state_` attribute to indicate the situation in which the graphic is used.

During each state change, Android traverses the state list from top to bottom. The first item that matches the current state is used, which means that the selection is not based on the "best match," but is simply the first item that meets the minimum criteria of the state.

The state list in the following example defines which image is shown for a button when the button is in different states. When the button is pressed—that is, when `state_pressed="true"` —the app shows an image named button_pressed. When the button is in focus (`state_focused="true"`), or when the button is being hovered over (`state_hovered="true"`), the app shows different buttons.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /> <!-- pressed -->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /> <!-- focused -->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /> <!-- hovered -->
    <item android:drawable="@drawable/button_normal" /> <!-- default -->
</selector>
```

Other available states include `android:state_selected,` `android:state_checkable, android:state_checked` , and others. For details about all the options, see the state list guide.

Level list drawables

A *level list drawable* defines alternate drawables, each assigned a maximum numerical value. To select which drawable to use, call the `setLevel()` method, passing in an integer that is matched against the maximum level integer defined in XML. The resource with the lowest maximum level greater than or equal to the integer passed into `setLevel()` is selected.

For example, the following XML defines a level list that includes two alternate drawables, `status_off` and `status_on` :

```
<?xml version="1.0" encoding="utf-8"?>
<level-list xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:drawable="@drawable/status_off"
        android:maxLevel="0" />
    <item
        android:drawable="@drawable/status_on"
        android:maxLevel="1" />
</level-list>
```

To select the `status_off` drawable, call `setLevel(0)` . To select the `status_on` drawable, call `setLevel(1)` .

An example use of a `LevelListDrawable` is a battery level indicator icon that uses different images to indicate different current battery levels.

Transition drawables

A `TransitionDrawable` is a drawable that cross-fades between two other drawables. To define a transition drawable in XML, use the `<transition>` element. Each drawable is represented by an `<item>` element inside the `<transition>` element. No more than two `<item>` elements are supported.

For example, this drawable cross-fades between an "on" state and an "off" state drawable:

```
<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/on" />
    <item android:drawable="@drawable/off" />
</transition>
```

To transition forward, meaning to shift from the first drawable to the second, call `startTransition()`. To transition in the other direction, call `reverseTransition()`. Each of these methods takes an argument of type `int`, representing the number of milliseconds for the transition.

Vector drawables

In Android 5.0 (API Level 21) and above, you can define *vector drawables*, which are images that are defined by a path. Vector drawables scale without losing definition. Most vector drawables use SVG files, which are plain text files or compressed binary files that include two-dimensional coordinates for how the image is drawn on the screen.

Because SVG files are text, they are more space efficient than most other image files. Also, you only need one file for a vector image instead of a file for each screen density, as is the case for bitmap images.

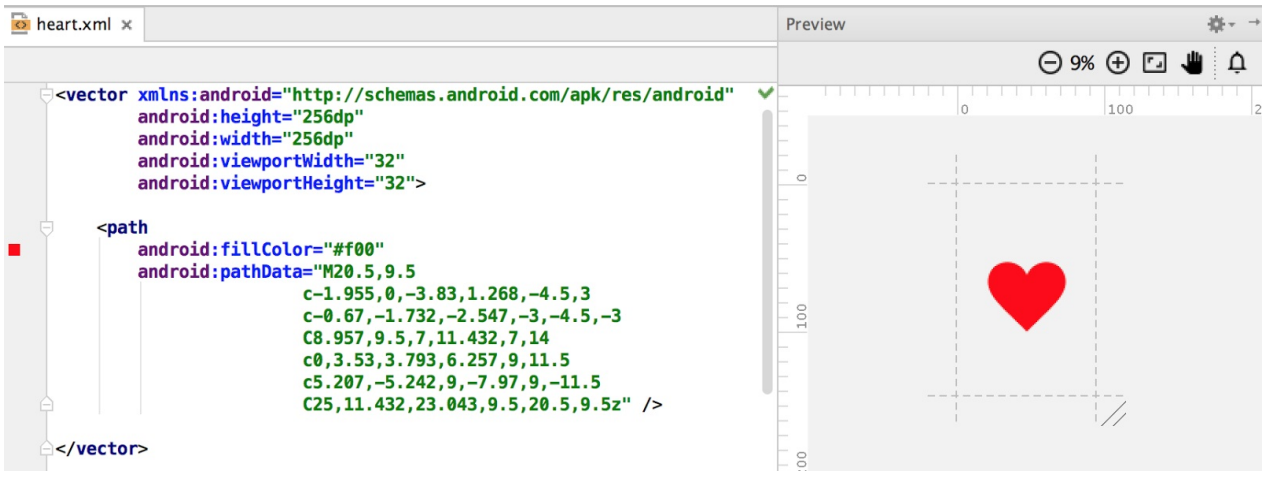
To bring an existing vector image or a Material Design icon into your Android Studio project as a vector drawable:

1. Right-click on the res/drawable folder.
2. Select **New > Vector Asset**. The Vector Asset Studio opens and guides you through the process.

To create a vector image, define the details of the shape inside a `<vector>` XML element. For example, the following code defines the shape of a heart and fills it with a red color (`#f00`):

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    <!-- intrinsic size of the drawable -->
    android:height="256dp"
    android:width="256dp"
    <!-- size of the virtual canvas -->
    android:viewportWidth="32"
    android:viewportHeight="32">

  <!-- draw a path -->
  <path android:fillColor="#f00"
      android:pathData="M20.5,9.5
                        c-1.955,0,-3.83,1.268,-4.5,3
                        c-0.67,-1.732,-2.547,-3,-4.5,-3
                        C8.957,9.5,7,11.432,7,14
                        c0,3.53,3.793,6.257,9,11.5
                        c5.207,-5.242,9,-7.97,9,-11.5
                        C25,11.432,23.043,9.5,20.5,9.5z" />
</vector>
```

Android Studio shows a preview of vector drawables, for example, here's the result of creating the XML file described above:



If you already have an image in SVG format, there are several ways to get the image's `pathData` information:

- In Android Studio, right-click on the drawable folder and select **New > Vector Asset** to open the Vector Asset Studio tool. Use the tool to import a local SVG file.
- Use a file-conversion tool such as svg2android.
- Open the image in a text editor, or if you're viewing the image in a browser, view the page source. Look for the `d=` information, which is equivalent to the `pathData` in your XML.

Vector images are represented in Android as `VectorDrawable` objects. For details about the `pathData` syntax, see the SVG Path reference. To learn how to animate the properties of vector drawables, see Animate Vector Drawables.

Images

Images, from launcher icons to banner images, are used in many ways in Android. Each use case has different requirements for image resolution, scalability and simplicity. In this section you learn about the different ways to generate images and include them in your app.

Creating icons

Every app requires at least a launcher icon, and apps often include icons for action bar actions, notifications, and other use cases.

There are two approaches to creating icons:

- Create a set of image files of the same icon in different resolutions and sizes so that the icon looks the same across devices with different screen densities. You can use Image Asset Studio to do this.
- Use vector drawables, which scale automatically without the image becoming pixelated or blurry. You can use Vector Asset Studio to do this.

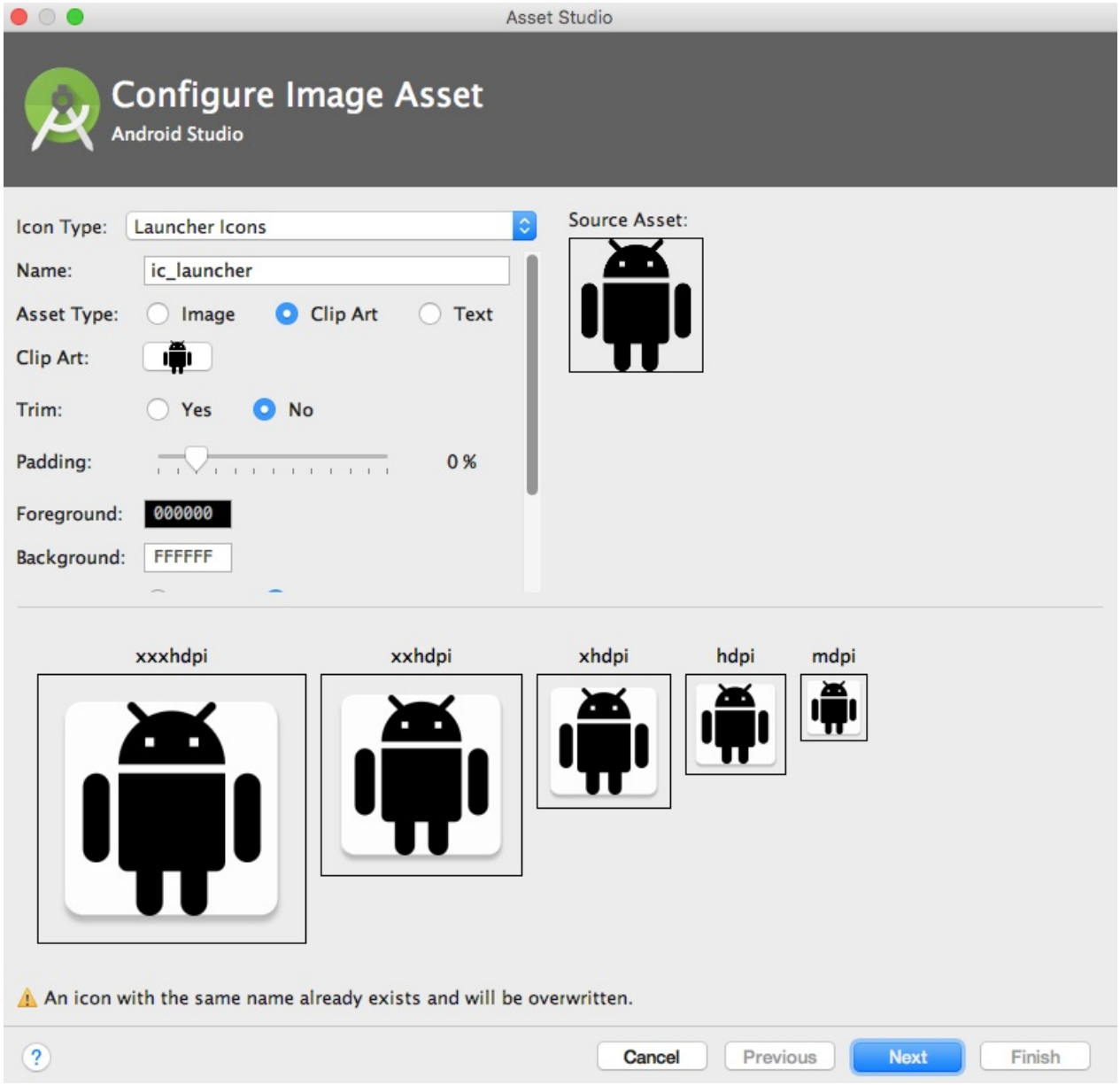
Image Asset Studio

Android Studio includes a tool called Image Asset Studio that helps you generate your own app icons from Material Design icons, custom images, and text strings. It generates a set of icons at the appropriate resolution for each generalized screen density that your app supports. Image Asset Studio places the newly generated icons in density-specific folders under the res/ folder in your project. At runtime, Android uses the appropriate resource based on the screen density of the device your app is running on.

Image Asset Studio helps you generate the following icon types:

- Launcher icons
- Action bar and tab icons
- Notification icons

To use Image Asset Studio, right-click on the res/ folder in Android Studio and select **New > Image Asset**. The Configure Asset Studio wizard opens and guides you through the process.



For more about Image Asset Studio, see the Image Asset Studio guide.

Vector Asset Studio

Starting with API 21, you can use vector drawables instead of image files for your icons.

Advantages of using vector drawables as icons:

- Vector drawables can reduce your APK file size dramatically, because you don't have to include multiple versions of each icon image. You can use one vector image to scale seamlessly to any resolution.
- Users might be more likely to download an app that has smaller files and a smaller package size.

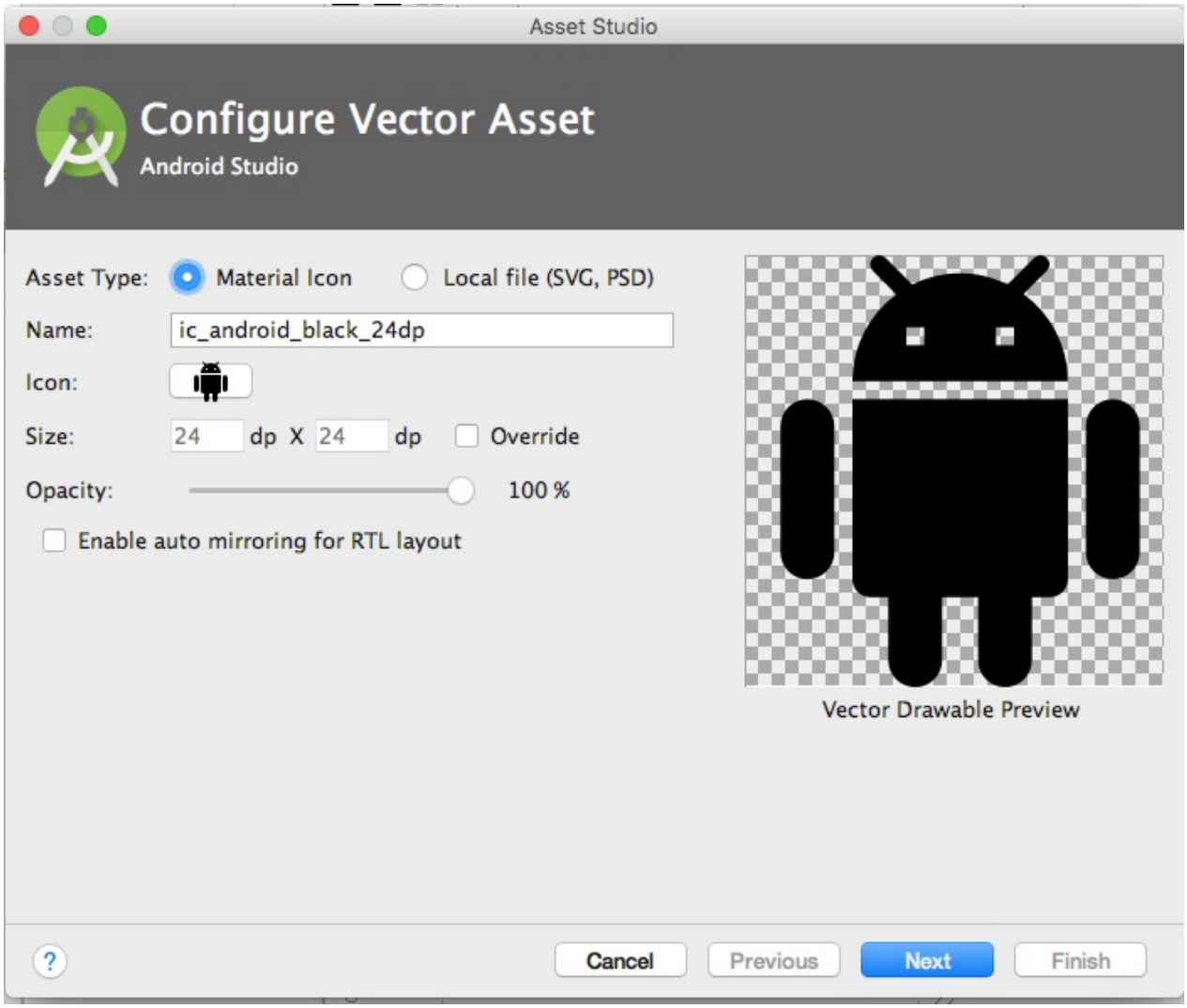
Disadvantages of using vector drawables as icons:

- A vector drawable can include only a limited amount of detail. Vector drawables are mostly used for less detailed icons such as the Material Design icons. Icons with more detail usually need image files.
- Vector drawables are not supported on devices running API level 20 or below.

To use vector drawables on devices running API level 20 or below, you have to decide between two methods of backward-compatibility:

- By default, at build time the system creates bitmap versions of your vector drawables in different resolutions. This allows the icons to run on devices that aren't able to draw vector drawables.
- The `VectorDrawableCompat` class in the Android Support Library allows you to support vector drawables in Android 2.1 (API level 7) and higher.

Vector Asset Studio is a tool that helps you add Material Design icons and vector drawables to your Android project. To use it, right-click on the res/ folder in Android Studio and select **New > Vector Asset**. The Configure Asset Studio wizard opens and guides you through the process.



For more information on using the Vector Asset Studio and supporting backward compatibility, refer to the Vector Asset Studio guide.

Creating other images

Banner images, user profile pictures, and other images come in all shapes and sizes. In many cases they are larger than they need to be for a typical application user interface (UI). For example, the system Gallery app displays photos taken using an Android device's camera, and these photos are typically much higher resolution than the screen density of the device. Android devices have finite memory, so ideally, you want to load only a lower resolution version of a photo in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution doesn't provide any visible benefit, but still takes up precious memory and adds additional performance overhead due to additional on-the-fly scaling.

You can load resized images manually, but several third party libraries have been created to help with loading, scaling and caching images.

Using image-loading libraries

Image-loading libraries like Glide and Picasso can handle image sizing, caching, and display. These third-party libraries are optimized for mobile, and they are well-documented.

Glide supports fetching, decoding, and displaying video stills, images, and animated GIFs. You can use Glide to load images from Web APIs, as well as ones located in your resource files. Glide includes features such as loading placeholder images (for loading more detailed images), cross-fade animations, and automatic caching.

To use Glide:

1. Download the library.
2. Include the dependency in your app-level build.gradle file, replacing `<em>n.n.n</em>` with the latest version of Glide: `compile 'com.github.bumptech.glide:glide:<em>n.n.n</em>'`

You can use Glide to load any image into a UI element. The following example loads an image from a URL into an `ImageView`:

```
ImageView imageView = (ImageView) findViewById(R.id.my_image_view);
Glide.with(this).load("URL").into(imageView);
```

In the code snippet, `this` refers to the context of the application. Replace "URL" with the URL of the image's location. By default, the image is stored in a local cache and accessed from there the next time it's called.

- For more examples, see the Glide wiki.
- For more about Glide, see the Glide documentation and the [glide] tag on stack overflow.
- For more about Picasso, see the Picasso documentation and the [picasso] tag on stack overflow.
- To compare the features of different libraries, search on stack overflow, for example Glide vs. Picasso.

Testing image rendering

Images render differently on different devices. To avoid surprises, use the Android Virtual Device (AVD) manager to create virtual devices that simulate screens of different sizes and densities. Use these AVDs to test all your images.

Speeding up your app

Fetching and caching images

When your app fetches an image, it can use a lot of data. To conserve data, make sure your request starts out as small as possible. Define and store pre-sized images on the server side, then request images that are already sized to the View.

Cache your images so that each image only needs to travel over the network once. When an image is requested, check your cache first. Only request the image over the network if the image is not in the cache. Use an image-loading library like Glide or Picasso to handle caching. These libraries also manage the size of the cache, getting rid of old or unused images. For more about libraries, see the libraries section of this chapter.

To maximize performance in different contexts, set conditional rules for how your app handles images, depending on connection type and stability. Use `ConnectivityManager` to determine the connection type and status, then set conditional rules accordingly. For example, when a user is on a data connection (not WiFi), downgrade the requested image resolution to less than screen resolution. Upgrade the requested screen resolution again when the user is on WiFi.

When your app is fetching images over a network, a slow connection might leave your user waiting. Here are ways to keep your app feeling fast, even if images load slowly:

- Prioritize more important images so that they load first. Libraries like Glide and Picasso let you order requests by image priority.
- Prioritize requests for text before requests for images. If your app is usable without images, for example if it's a news feed app, letting a user scroll past your image can make the app functional and might even render the image request

obsolete.
- Display placeholder colors while fetching images.

If you display placeholder colors, you want the look of your app to stay consistent while the app loads images. Use the Palette library to select a placeholder color based on the requested image's color balance. First, include the Palette library in your build.gradle file:

```
dependencies: {
    compile 'com.android.support:palette-v7:24.2.1'
}
```

Pull the dominant color for the image you want and set it as the background color in your `ImageView`. If you fetch the image using a library, put the following code after you've defined the URL to load into the `ImageView`:

```
Palette palette = Palette.from(tiles).generate(new PaletteAsyncListener(){
   Public void onGenerated(Pallet pallette) {
      Palette.Swatch background = palette.getDominantSwatch();
         if (background != null) {
            ImageView.setBackgroundColor(background.getRgb());
          }
     }
}
```

Serving images over a network

To save bandwidth and keep your app moving fast, use WebP formats to serve and send images.

Another way to save bandwidth is to serve and cache custom-sized images. To do this, allow clients to specify the resolution and size required for their device and View, then generate and cache the needed image on the server side before you send it.

For example, a news feed landing page might request only a thumbnail image. Instead of sending a full-sized image, send only the thumbnail specified by that `ImageView`. You can further reduce the size of the thumbnail by producing images at different resolutions.

Tip: Use the `Activity.isLowRamDevice()` method to find out whether a device defines itself as "low RAM." If the method returns `true`, send low-resolution images so that your app uses less on-device memory.

Styles

In Android, a *style* is a collection of attributes that define the look and format of a View. You can apply the same style to any number of Views in your app; for example, several TextViews might have the same text size and layout. Using styles allows you to keep these common attributes in one location and apply them to each TextView using a single line of code in XML.

You can define styles yourself or use one of the platform styles that Android provides.

Defining and applying styles

To create a style, add a `<style>` element inside a `<resources>` element in any XML file located in the res/values/ folder. When you create a project in Android Studio, a res/values/styles.xml file is created for you.

A `<style>` element includes the following:

- A `name` attribute. Use the style's name when you apply the style to a View.
- An optional `parent` attribute. You learn about using `parent` attributes in the Inheritance section below.
- Any number of `<item>` elements as child elements of `<style>`. Each `<item>` element includes one style attribute.

This example creates a style that formats text to use a light gray monospace typeface so it looks like code:

```
<resources>
    <style name="CodeFont">
        <item name="android:typeface">monospace</item>
        <item name="android:textColor">#D7D6D7</item>
    </style>
</resources>
```

The following XML applies the new `CodeFont` style to a View:

```
<TextView
    style="@style/CodeFont"
    android:text="@string/code_string" />
```

Inheritance

A new style can inherit the properties of an existing style. When you create a style that inherits properties, you define only the properties that you want to change or add. You can inherit properties from platform styles and from styles that you create yourself. **To inherit a platform style,** use the `parent` attribute to specify the resource ID of the style you want to inherit. For example, here's how to inherit the Android platform's default text appearance (the `TextAppearance` style) and change its color:

```
<style name="GreenText" parent="@android:style/TextAppearance">
    <item name="android:textColor">#00FF00</item>
</style>
```

To apply this style, use `@style/GreenText`. **To inherit a style that you created yourself,** use the name of the style you want to inherit as the first part of the new style's name, and separate the parts with a period:

```
 name="StyleToInherit.Qualifier"
```

For example, to create a style that inherits the `CodeFont` style defined above, use `CodeFont` as the first part of the new style's name:

```
<style name="CodeFont.RedLarge">
    <item name="android:textColor">#FF0000</item>
    <item name="android:textSize">34sp</item>
</style>
```

This example includes the `typeface` attribute from the original `CodeFont` style, overrides the original `textColor` attribute with red, and adds a new attribute, `textSize`. To apply this style, use `@style/CodeFont.RedLarge`.

Themes

You create a theme the same way you create a style, which is by adding a `<style>` element inside a `<resources>` element in any XML file located in the res/values/ folder.

What's the difference between a style and a theme?

- A *style* applies to a View. In XML, you apply a style using the `style` attribute.
- A *theme* applies to an entire Activity or application, rather than to an individual View. In XML, you apply a theme using the `android:theme` attribute.

Any style can be used as a theme. For example, you could apply the `CodeFont` style as a theme for an Activity, and all the text inside the Activity would use gray monospace font.

Applying themes

To apply a theme to your app, declare it inside an `<application>` element inside the AndroidManifest.xml file. This example applies the `AppTheme` theme to the entire application:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.exampledomain.myapp">
    <application
        ...
        android:theme="@style/AppTheme">
    </application>
    ...
```

To apply a theme to an Activity, declare it inside an `<activity>` element in the AndroidManifest.xml file. In this example, the `android:theme` attribute applies the Theme_Dialog platform theme to the Activity:

```
<activity android:theme="@android:style/Theme.Dialog">
```

Default theme

When you create a new project in Android Studio, a default theme is defined for you within the styles.xml file. For example, this code might be in your styles.xml file:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
   <!-- Customize your theme here. -->
   <item name="colorPrimary">@color/colorPrimary</item>
   <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
   <item name="colorAccent">@color/colorAccent</item>
</style>
```

In this example, `AppTheme` inherits from `Theme.AppCompat.Light.DarkActionBar`, which is one of the many Android platform themes available to you. (You'll learn about the color attributes in the unit on Material Design.)

Platform styles and themes

The Android platform provides a collection of styles and themes that you can use in your app. To find a list of all of them, you need to look in two places:

- The R.style class lists most of the available platform styles and themes.
- The support.v7.appcompat.R.style class lists more of them. These styles and themes have "`AppCompat`" in their names, and they are supported by the v7 appcompat library.

The style and theme names include underscores. To use them in your code, replace the underscores with periods. For example, here's how to apply the Theme_NoTitleBar theme to an activity:

```
<activity android:theme="@android:style/Theme.NoTitleBar"
```

And here's how to apply the AlertDialog_AppCompat style to a View:

```
<TextView
    style="@style/AlertDialog.AppCompat"
    android:text="@string/code_string" />
```

The documentation doesn't describe all the styles and themes in detail, but you can infer things about them from their names. For example, in `Theme.AppCompat.Light.DarkActionBar`

- "Theme" indicates that this style is meant to be used as a theme.
- "AppCompat" indicates that this theme is supported by the v7 appcompat library.
- "Light" indicates that the theme consists of light background, white by default. All the text colors in this theme are dark,

to contrast with the light background. (If you wanted a dark background and light text, your theme could inherit from a theme such as `Theme.AppCompat` without "Light" in the name.)
- "DarkActionBar" indicates that a dark color is used for the action bar, so any text or icons in the action bar are a light color.

Another useful theme is `Theme.AppCompat.DayNight`, which enables the user to browse in a low-contrast "night mode" at night. It automatically changes the theme from `Theme.AppCompat.Light` to `Theme.AppCompat`, based on the time of day. To learn more about the `DayNight` theme, read Chris Banes's blog post.

To learn more about using platform styles and themes, visit the styles and themes guide.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Drawables, Styles, and Themes

Learn more

- Drawable resources guide
- Image Asset Studio guide
- Vector Asset Studio guide
- Roman Nurik's Android Asset Studio
- Styles and themes guide

5.2: Material Design

Table of Contents:

- Introduction
- Principles of Material Design
- Colors
- Typography
- Layout
- Components and patterns
- Motion
- Animations
- Related practical
- Learn more

Material Design is a visual design philosophy that Google created in 2014. The aim of Material Design is a unified user experience across platforms and device sizes. Material Design includes a set of guidelines for style, layout, motion, and other aspects of app design. The complete guidelines are available in the Material Design Spec.

Material Design is for desktop web applications as well as for mobile apps. This chapter focuses only on Material Design for mobile apps on Android.

Principles of Material Design

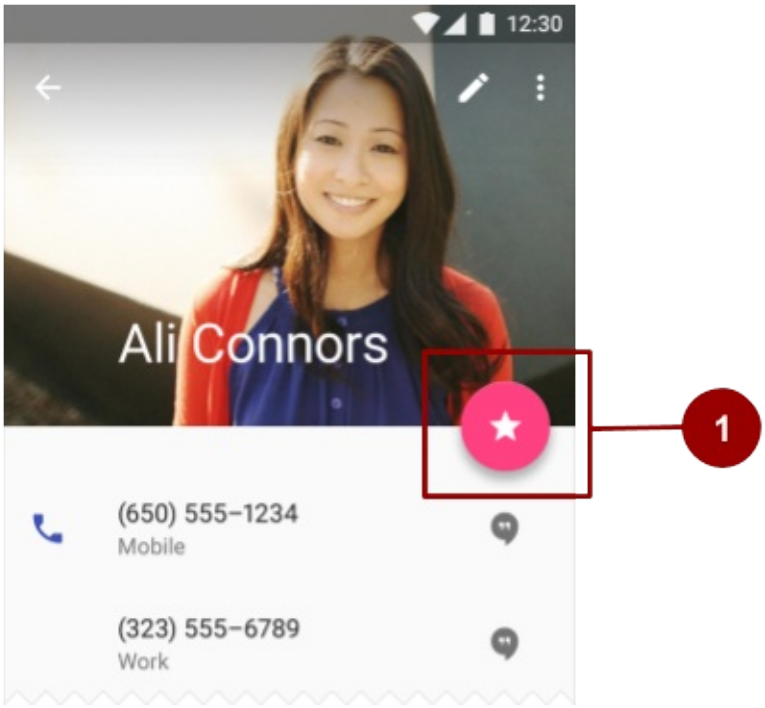
The "material" metaphor

In Material Design, elements in your Android app behave like real world materials: they cast shadows, occupy space, and interact with each other.

Bold, graphic, intentional

Material Design involves deliberate color choices, edge-to-edge imagery, large-scale typography, and intentional white space that create a bold and graphic interface.

Emphasize user actions in your app so that the user knows right away what to do, and how to do it. For example, highlight things that users can interact with, such as buttons, EditText fields, and switches.



1. In this layout, the floating action button is highlighted with a pink accent color.

Meaningful motion

Make animations and other motions in your app meaningful, so they don't happen at random. Use motions to reinforce the idea that the user is the app's primary mover. For example, design your app so that most motions are initiated by the user's actions, not by events outside the user's control. You can also use motion to focus the user's attention, give the user subtle feedback, or highlight an element of your app.

When your app presents an object to the user, make sure the motion doesn't break the continuity of the user's experience. For example, the user shouldn't have to wait for an animation or transition to complete.

The Motion section in this chapter goes into more detail about how to use motion in your app.

Colors

Material Design color palette

Material Design principles include the use of bold color. The Material Design color palette contains colors to choose from, each with a primary color and shades labeled from 50 to 900:

- Choose a color labeled "500" as the primary color for your brand. Use that color and shades of that color in your app.
- Choose a contrasting color as your accent color and use it to create highlights in your app. Select any color that starts with "A."

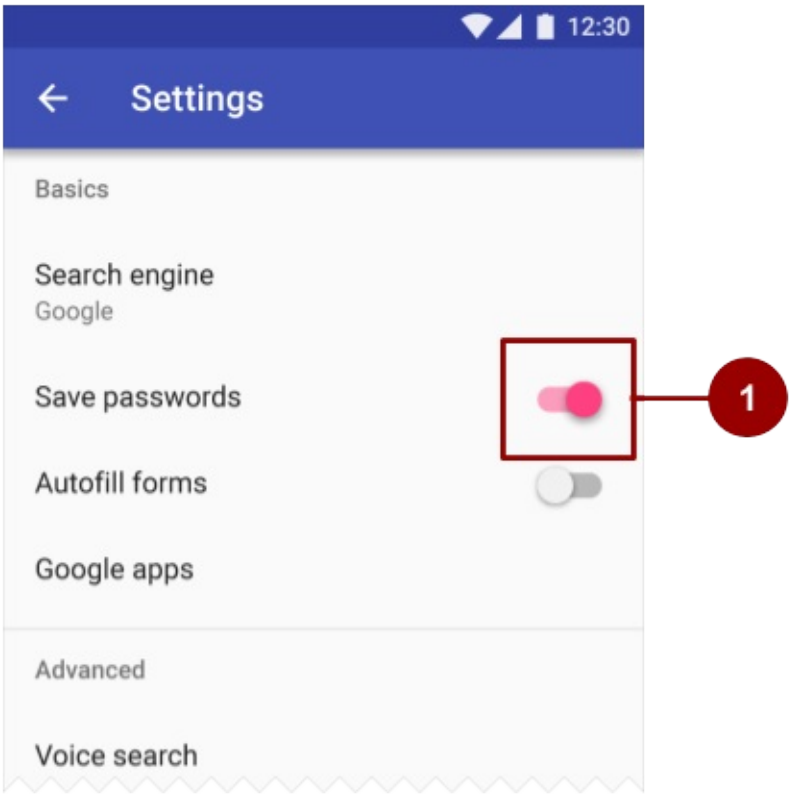
When you create an Android project in Android Studio, a sample Material Design color scheme is selected for you and applied to your theme. In values/colors.xml, three `<color>` elements are defined, `colorPrimary`, `colorPrimaryDark`, and `colorAccent`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <!-- Indigo. -->
    <color name="colorPrimaryDark">#303F9F</color>
    <!-- A darker shade of indigo. -->
    <color name="colorAccent">#FF4081</color>
    <!-- A shade of pink. -->
</resources>
```

In values/styles.xml, the three defined colors are applied to the default theme, which applies the colors to some app elements by default:

- `colorPrimary` is used by several Views by default. For example, in the `AppTheme` theme, `colorPrimary` is used as the background color for the action bar. Change this value to the "500" color that you select as your brand's primary color.
- `colorPrimaryDark` is used in areas that need to slightly contrast with your primary color, for example the status bar. Set this value to a slightly darker version of your primary color.
- `colorAccent` is used as the highlight color for several Views. It's also used for switches in the "on" position, floating action buttons, and more.

In the screenshot below, the background of the action bar uses `colorPrimary` (indigo), the status bar uses `colorPrimaryDark` (a darker shade of indigo), and the switch in the "on" position uses `colorAccent` (a shade of pink).



1. In this layout, the switch in the "on" position is highlighted with a pink accent color.

In summary, here's how to use the Material Design color palette in your Android app:

1. Pick a primary color for your app from Material Design color palette and copy its hex value into the `colorPrimary` item in colors.xml.
2. Pick a darker shade of this color and copy its hex value into the `colorPrimaryDark` item.
3. Pick an accent color from the shades starting with an "A" and copy its hex value into the colorAccent item.
4. If you need more colors, create additional `<color>` elements in the colors.xml file. For example, you could pick a lighter version of indigo and create an additional `<color>` element named `colorPrimaryLight`. (The name is up to you.)

```
<color name="colorPrimaryLight">#9FA8DA</color>
<!-- A lighter shade of indigo. -->
```

To use this color, reference it as `@color/colorPrimaryLight`.

Changing the values in colors.xml automatically changes the colors of the Views in your app, because the colors are applied to the theme in styles.xml.

Contrast

Make sure all the text in your app's UI contrasts with its background. Where you have a dark background, make the text on top of it a light color, and vice versa. This kind of contrast is important for readability and accessibility, because not all people see colors the same way.

If you use a platform theme such as `Theme.AppCompat`, contrast between text and its background is handled for you. For example:

- If your theme inherits from `Theme.AppCompat`, the system assumes you are using a dark background. Therefore all of the text is near white by default.
- If your theme inherits from `Theme.AppCompat.Light`, the text is near black, because the theme has a light background.
- If you use the `Theme.AppCompat.Light.DarkActionBar` theme, the text in the action bar is near white, to contrast with the action bar's dark background. The rest of the text in the app is near black, to contrast with the light background.

Use color contrast to create visual separation among the elements in your app. Use your `colorAccent` color to to call attention to key UI elements such as floating action buttons and switches in the "on" position.

Opacity

Your app can display text with different degrees of opacity to convey the relative importance of information. For example, text that's less important might be nearly transparent (low opacity).

Set the `android:textColor` attribute using any of these formats: `"#rgb"`, `"#rrggbb"`, `"#argb"`, or `"#aarrggbb"`. To set the opacity of text, use the `"#argb"` or `"#aarrggbb"` format and include a value for the *alpha channel*. The alpha channel is the `a` or the `aa` at the start of the `textColor` value.

The maximum opacity value, `FF` in hex, makes the color completely opaque. The minimum value, `00` in hex, makes the color complete transparent.

To determine what hex number to use in the alpha channel:

1. Decide what level of opacity you want to use, as a percentage. The level of opacity used for text depends on whether your background is dark or light. To find out what level of opacity to use in different situations, see the Text color portion of the Material Design guide.
2. Multiply that percentage, as a decimal value, by 255. For example, if you need primary text that's 87% opaque, multiply 0.87 x 255. The result is 221.85.
3. Round the result to the nearest whole number: 222.
4. Use a hex converter to convert the result to hex: `DE`. If the result is a single value, prefix it with `0`.

In the following XML code, the background of the text is dark, and the color of the primary text is 87% white (`deffffff`). The first two numbers of the color code (`de`) indicate the opacity.

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello World!"
   android:textSize="45dp"
   android:background="@color/colorPrimaryDark"
   android:textColor="#deffffff"/>
```

Typography

Typeface

Roboto is the standard Material Design typeface on Android. Roboto has six weights: Thin, Light, Regular, Medium, Bold,

and Black.

Font styles

The Android platform provides predefined font styles and sizes that you can use in your app. These styles and sizes were developed to balance content density and reading comfort under typical conditions. Type sizes are specified with sp (scaleable pixels) to enable large type modes for accessibility.

Be careful not to use too many different type sizes and styles together in your layout.

To use one of these predefined styles in a View, set the `android:textAppearance` attribute. This attribute defines the default appearance of the text: its color, typeface, size, and style. Use the backward-compatible `TextAppearance.AppCompat` style.

For example, to make a TextView appear in the Display 3 style, add the following attribute to the TextView in XML:

```
android:textAppearance="@style/TextAppearance.AppCompat.Display3"
```

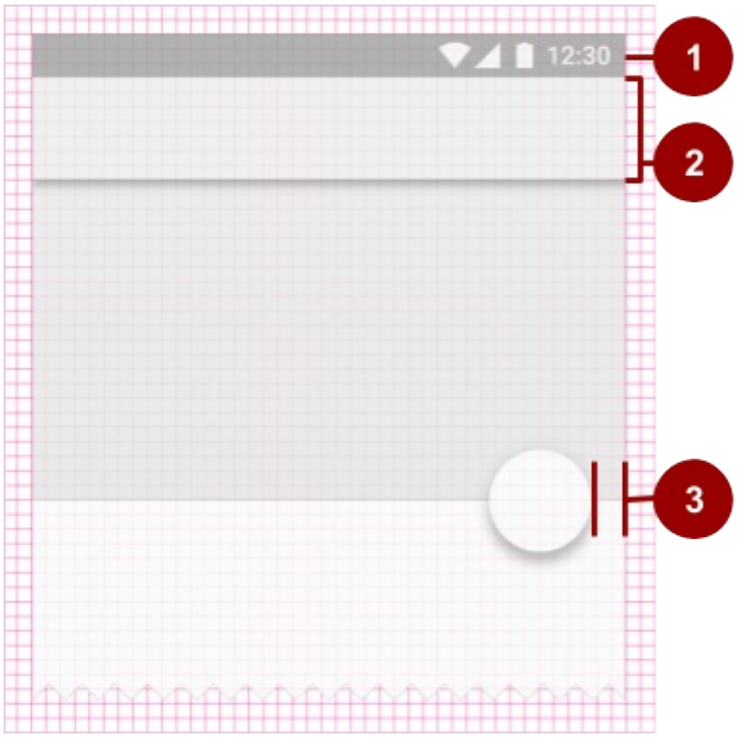
For more information on styling text, view the Typography Material Design guidelines.

Layout

Metrics and keylines

Components in the Material Design templates that are meant for mobile, tablet, and desktop devices align to an 8dp square grid. A *dp* is a density-independent pixel, an abstract unit based on screen density. A dp is similar to an sp, but sp is also scaled by the user's font size preference. That's why sp is preferred for accessibility. For more about units of measurement, refer to the Layouts, Views, and Resources unit.

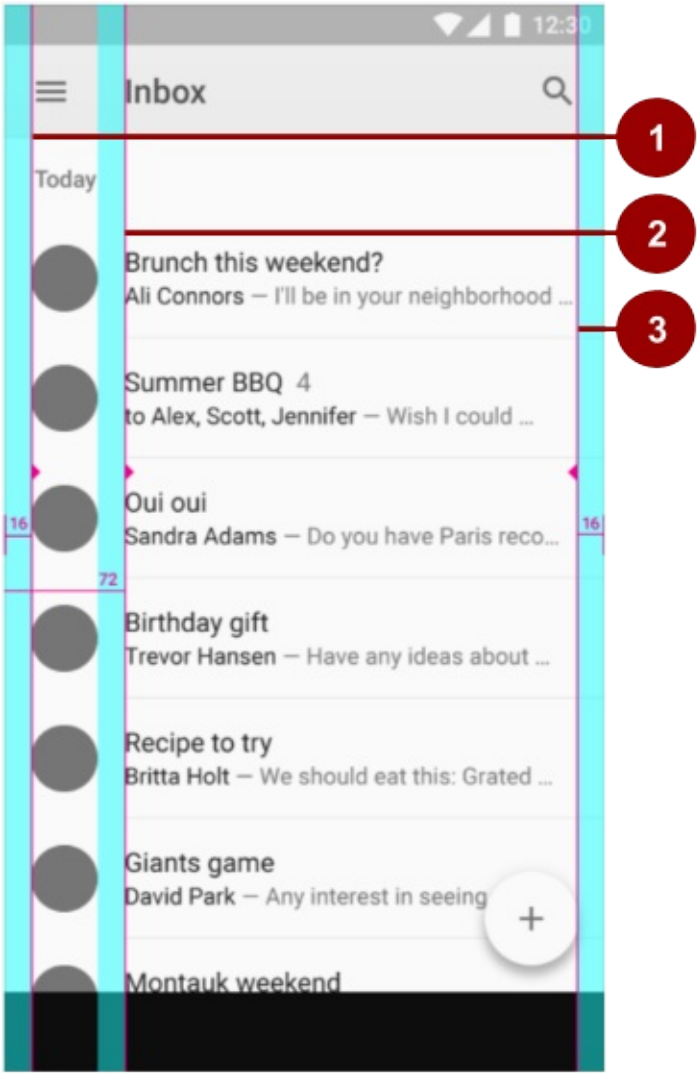
The 8dp square grid guides the placement of elements in your layout. Every square in the grid is 8dp x 8dp, so the height and width of every element in the layout is a multiple of 8dp.



1. The status bar in this layout is 24dp tall, the height of three grid squares.
2. The toolbar is 56dp tall, the height of seven grid squares.
3. One of the right-hand content margins is 16dp from the edge of the screen, the width of two grid squares.

Iconography in toolbars align to a 4dp square grid instead of an 8dp square grid, so the dimensions of icons in the toolbar are multiples of 4dp.

Keylines are outlines in a layout grid that determine the placement of text and icons. For example, keylines mark the edges of the margins in a layout.



1. Keyline showing the left margin for the screen edge, which in this case is 16dp.
2. Keyline showing the left margin for content associated with an icon or avatar, 72dp.
3. Keyline showing the right margin for the screen edge, 16dp.

Material Design typography aligns to a 4dp *baseline grid*, which is a grid made up only of horizontal lines.

The Material Design guide provides downloadable templates for commonly used UI screens. To learn more about metrics and keylines in Material Design, visit the metrics and keylines guide.

Components and patterns

Buttons and many other Views used in Android conform by default to Material Design principles. The Material Design guide includes *components and patterns* that you can build on to help your users intuit how the elements in your UI work, even if users are new to your app.

Use Material Design components to guide the specs and behavior of buttons, chips, cards, and many other UI elements. Use Material Design patterns to guide how you format dates and times, gestures, the navigation drawer, and many other aspects of your UI.

This section teaches you about the Design Support Library and some of the components and patterns that are available to you. For complete documentation about all the components and patterns that you can use, see the Material Design guide.

Design Support Library

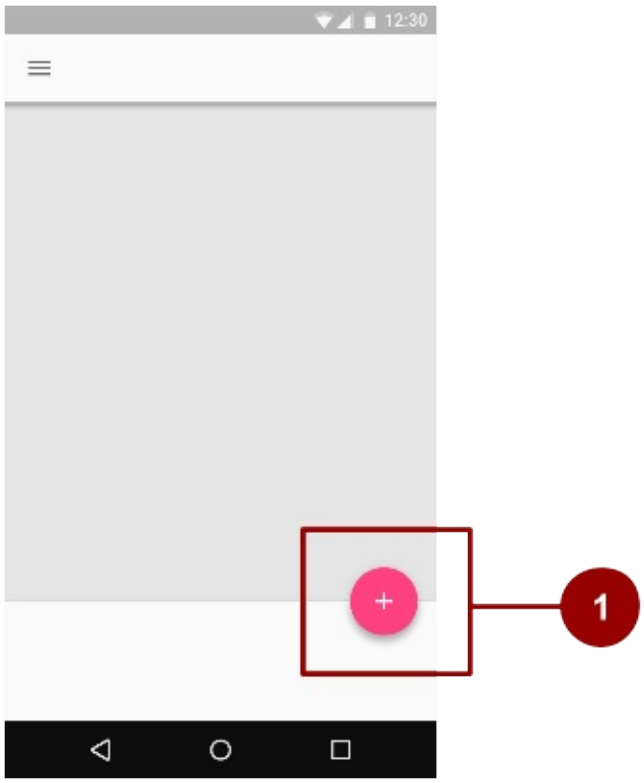
The Design package provides APIs to support adding Material Design components and patterns to your apps. The Design Support Library adds support for various Material Design components and patterns for you to build on. To use the library, include the following dependency in your build.gradle file:

```
compile 'com.android.support:design:25.0.1'
```

To make sure you have the most recent version number for the Design Support Library, check the Support Library page.

Floating action buttons (FABs)

Use a *floating action button (FAB)* for actions you want to encourage users to take. A FAB is a circled icon that floats "above" the UI. On focus it changes color slightly, and it appears to lift up when selected. When tapped, it can contain related actions.



1. A normal-sized FAB

To implement a FAB, use the `FloatingActionButton` widget and set the FAB's attributes in your layout XML. For example:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/addNewItemFAB"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_plus_sign"
    app:fabSize="normal"
    app:elevation="10%" />
```

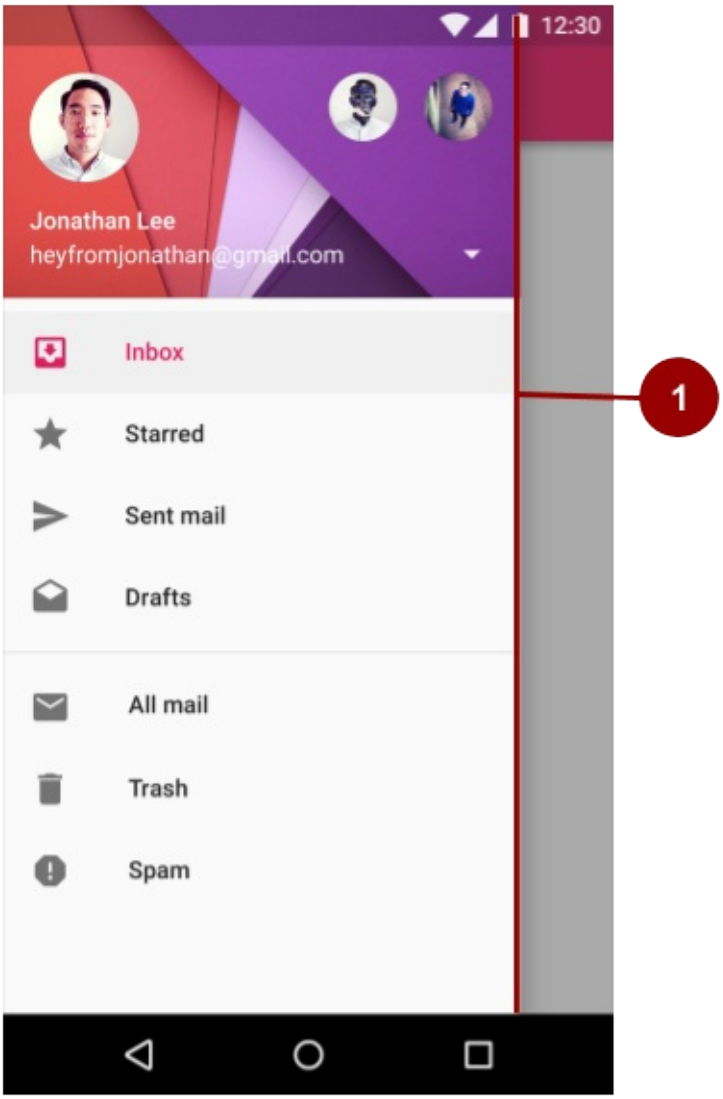
The `fabSize` attribute sets the FAB's size. It can be `"normal"` (56dp), `"mini"` (40dp), or `"auto"`, which changes based on the window size.

The FAB's *elevation* is the distance between its surface and the depth of its shadow. You can set the `elevation` attribute as a reference to another resource, a string, a Boolean, or several other ways.

To learn about all the attributes you can set for a FAB including `clickable`, `rippleColor`, and `backgroundTint`, see FloatingActionButton. To make sure you're using FABs as intended, check the extensive FAB usage information in the Material Design guide.

Navigation drawers

A *navigation drawer* is a panel that slides in from the left and contains navigation destinations for your app. A navigation drawer spans the height of the screen, and everything behind it is visible, but darkened.



1. An "open" navigation drawer

To implement a navigation drawer, use the `DrawerLayout` APIs available in the Support Library.

In your XML, use a `DrawerLayout` object as the root view of your layout. Inside it, add two views, one for your primary layout when the drawer is hidden, and one for the contents of the drawer.

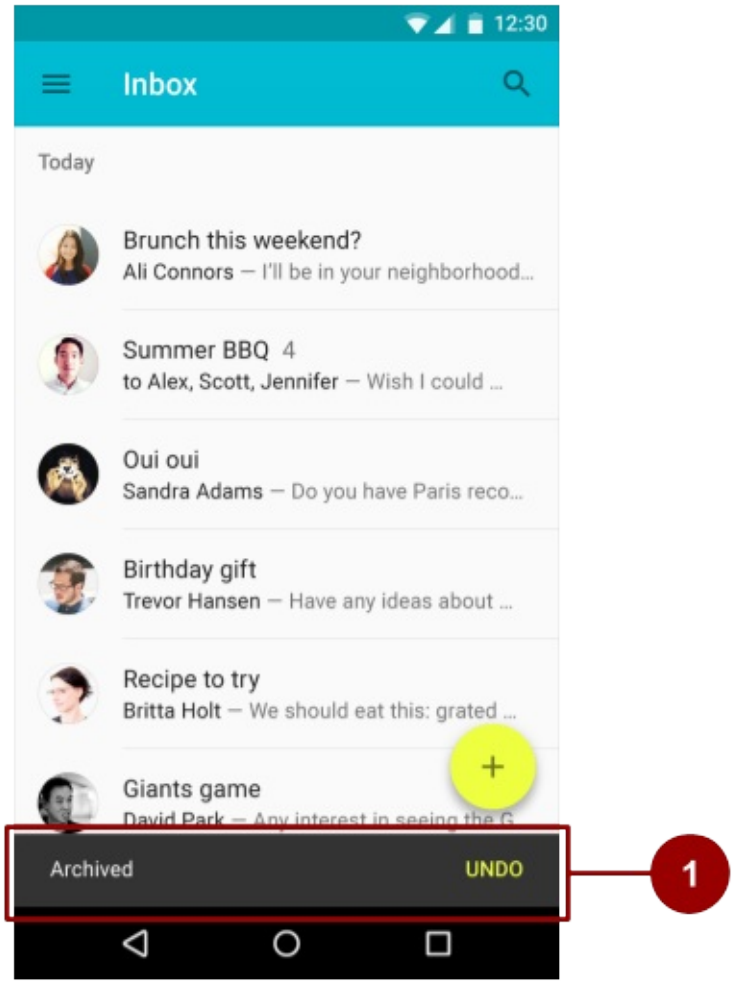
For example, the following layout has two child views: a `FrameLayout` to contain the main content (populated by a `Fragment` at runtime), and a `ListView` for the navigation drawer.

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- The main content view -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <!-- The navigation drawer -->
    <ListView android:id="@+id/left_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>
```

For more information, see Creating a Navigation Drawer and the usage information in the Material Design guide.

Snackbars

A *snackbar* provides brief feedback about an operation through a message in a horizontal bar on the screen. It contains a single line of text directly related to the operation performed. A snackbar can contain a text action, but no icons.



1. Snackbar

Snackbars automatically disappear after a timeout or after a user interaction elsewhere on the screen. You can associate a snackbar with any kind of view (any object derived from the `View` class). However, if you associate the snackbar with a `CoordinatorLayout`, the snackbar gains additional features:

- The user can dismiss the snackbar by swiping it away.
- The layout moves some other UI elements when the snackbar appears. For example, if the layout has a FAB, the layout moves the FAB up when it shows the snackbar, instead of drawing the snackbar on top of the FAB.

To create a `Snackbar` object, use the `Snackbar.make()` method. Specify the ID of the `CoordinatorLayout` view to use for the snackbar, the message that the snackbar displays, and the length of time to show the message. For example, this Java statement creates the snackbar and calls `show()` to show the snackbar to the user:

```
Snackbar.make(findViewById(R.id.myCoordinatorLayout), R.string.email_sent,
                        Snackbar.LENGTH_SHORT).show;
```

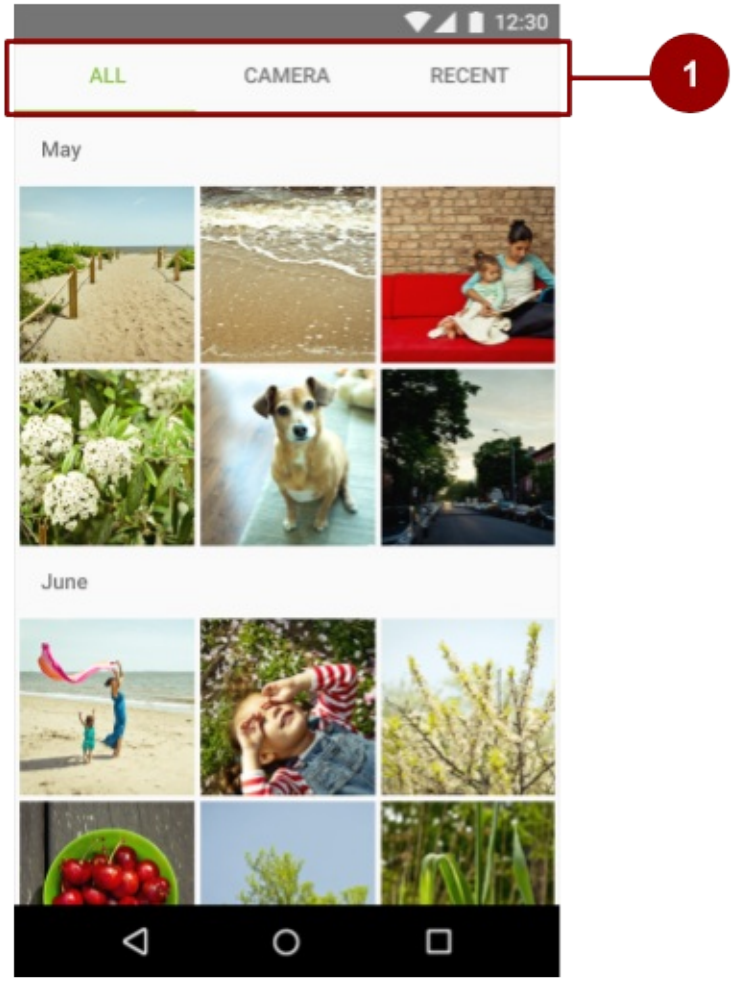
For more information, see Building and Displaying a Pop-Up Message and the Snackbar reference. To make sure you're using snackbars as intended, see the snackbar usage information in the Material Design guide.

Tip: A *toast* is similar to a snackbar, but toasts are usually used for system messaging, and toasts can't be swiped off the screen.

Tabs

Use *tabs* to organize content at a high level. For example, the user might use tabs to switch between Views, data sets, or functional aspects of an app. Present tabs as a single row above their associated content. Make tab labels short and informative.

You can you use tabs with *swipe views* in which users navigate between tabs with a horizontal finger gesture (horizontal paging). If your tabs use swipe views, don't pair the tabs with content that also supports swiping.



1. Three tabs, with the **ALL** tab selected

For information on implementing tabs, see Creating Swipe Views with Tabs. To make sure you're using tabs as intended, see the extensive tab usage information in the Material Design guide.

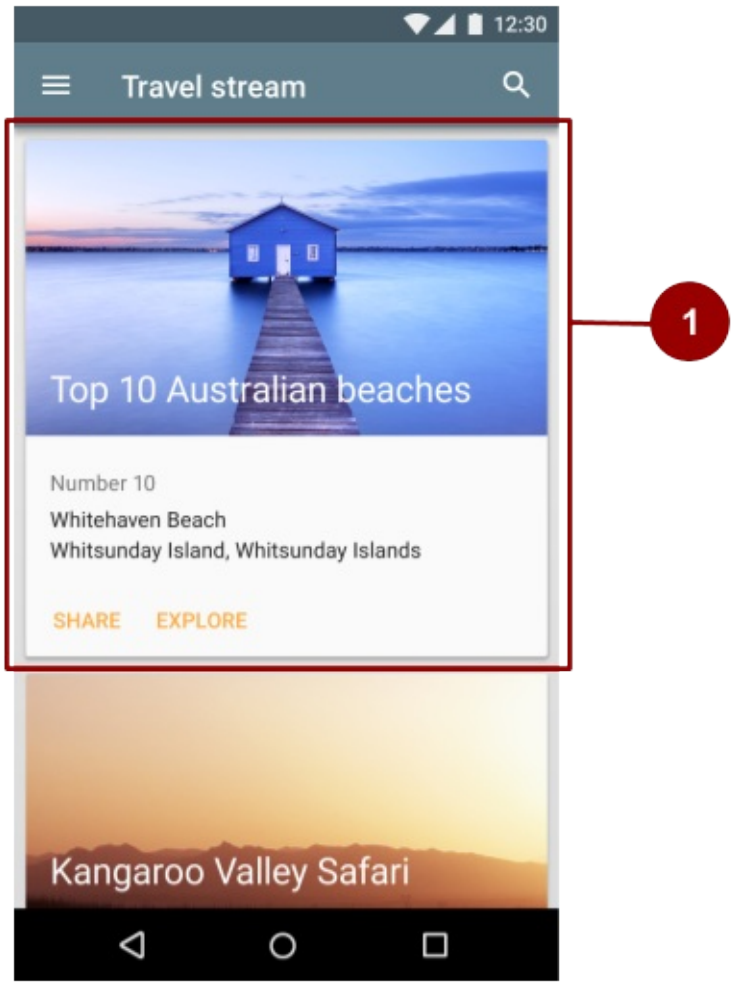
Cards

A *card* is a sheet of material that serves as an entry point to more detailed information. Each card covers only one subject. A card may contain a photo, text, and a link. It can display content containing elements of varying size, such as photos with captions of variable length.

A *card collection* is a layout of cards on the same plane.

The `CardView` widget is included as part of the v7 support library. To use it, add the following dependency to your build.gradle file:

```
compile 'com.android.support:cardview-v7:24.2.1'
```

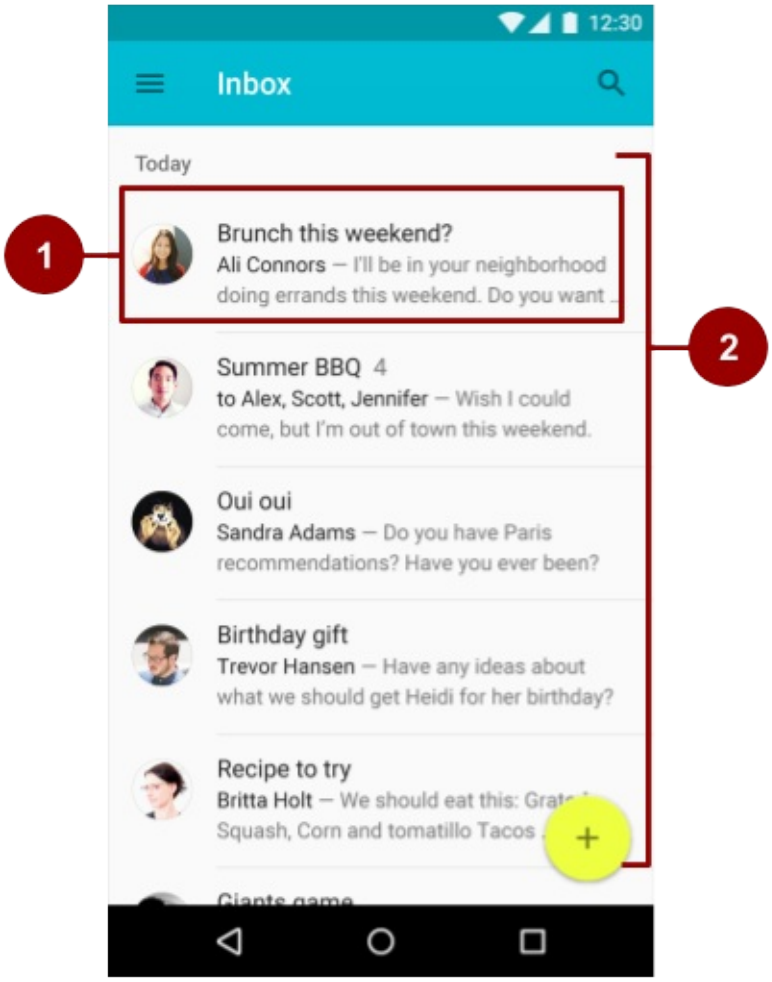


1. One card in a card collection

For more information on using the `CardView` widget, visit the card guide.

Lists

A *list* is a single continuous column of rows of equal width. Each row functions as a container for a tile. *Tiles* hold content,


12:30
Inbox
Today
Brunch this weekend?
Ali Connors — I'll be in your neighborhood doing errands this weekend. Do you want
Summer BBQ 4
to Alex, Scott, Jennifer — Wish I could come, but I'm out of town this weekend.
Oui oui
Sandra Adams — Do you have Paris recommendations? Have you ever been?
Birthday gift
Trevor Hansen — Have any ideas about what we should get Heidi for her birthday?
Recipe to try
Britta Holt — We should eat this:
Squash, Corn and tomatillo Tacos
1
2

and can vary in height within a list.

1. A tile within the list
2. A list with rows of equal width, each containing a tile

To create a list, use the `RecyclerView` widget. Include the following dependency in your build.gradle file.

```
compile 'com.android.support:recyclerview-v7:24.2.1'
```

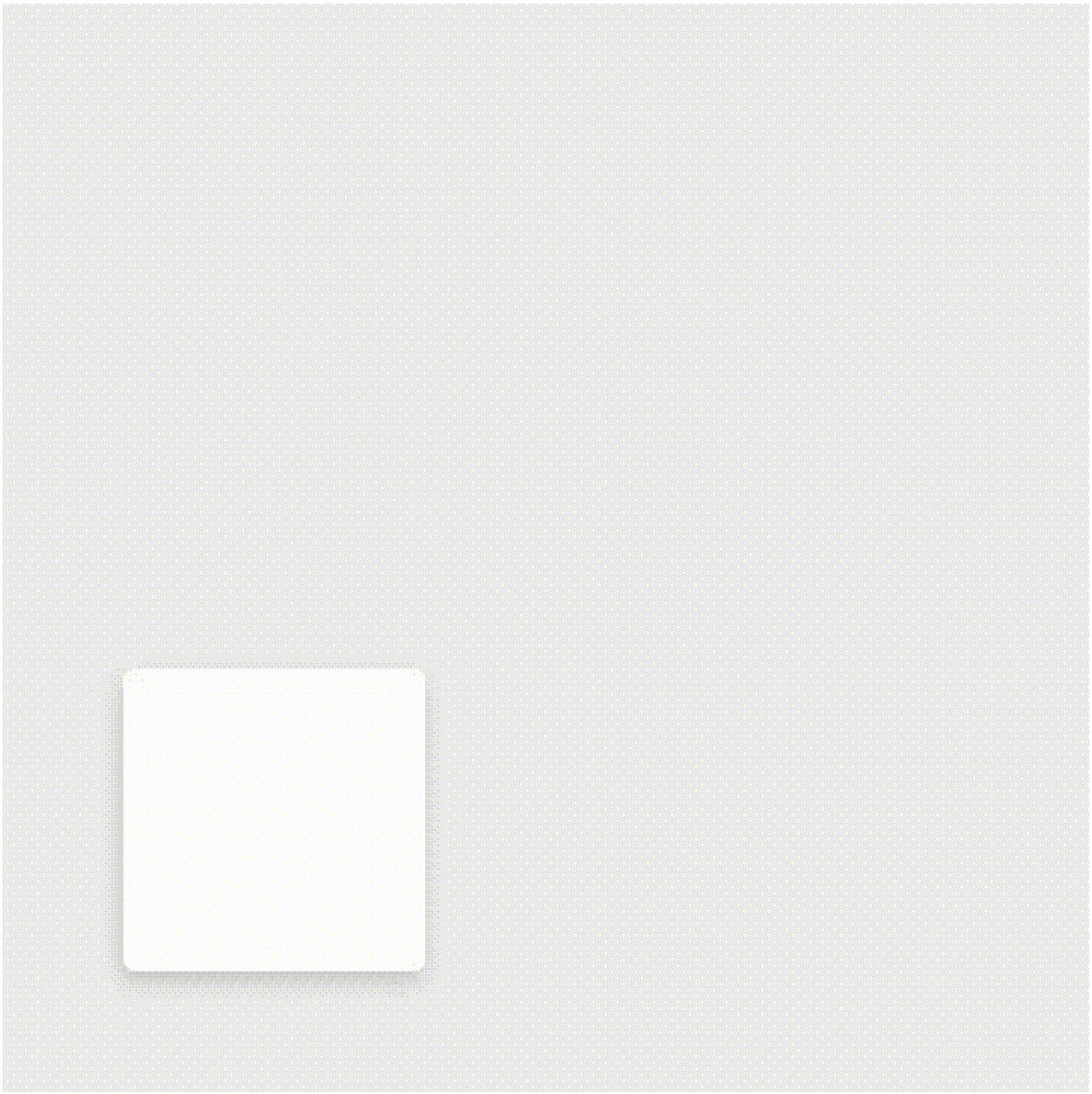
For more information on creating lists in Android, see the creating lists guide.

Motion

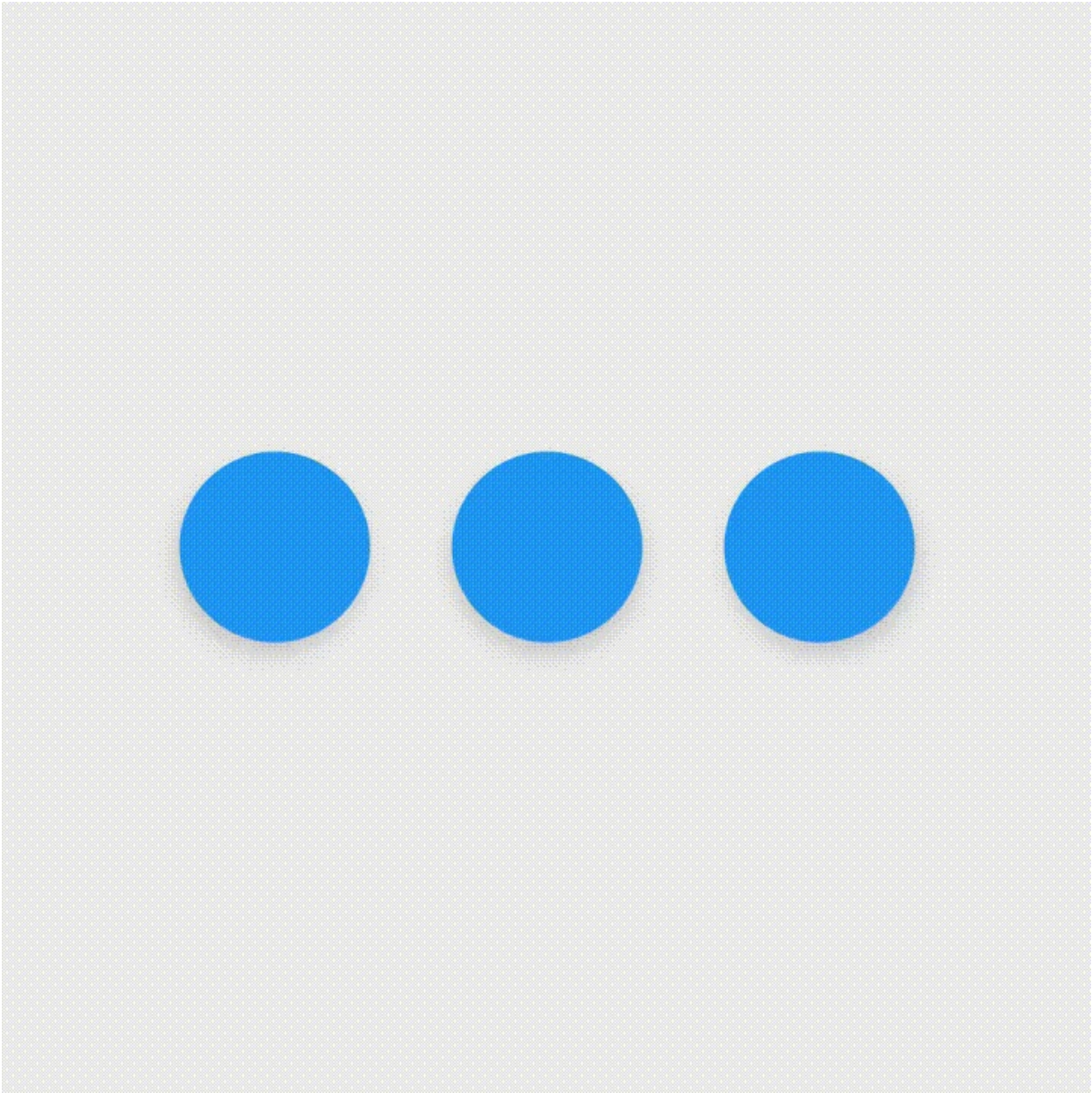
Motion in the world of Material Design is used to describe spatial relationships, functionality, and intention with beauty and fluidity. Motion shows how an app is organized and what it can do.

Motion in Material Design must be:

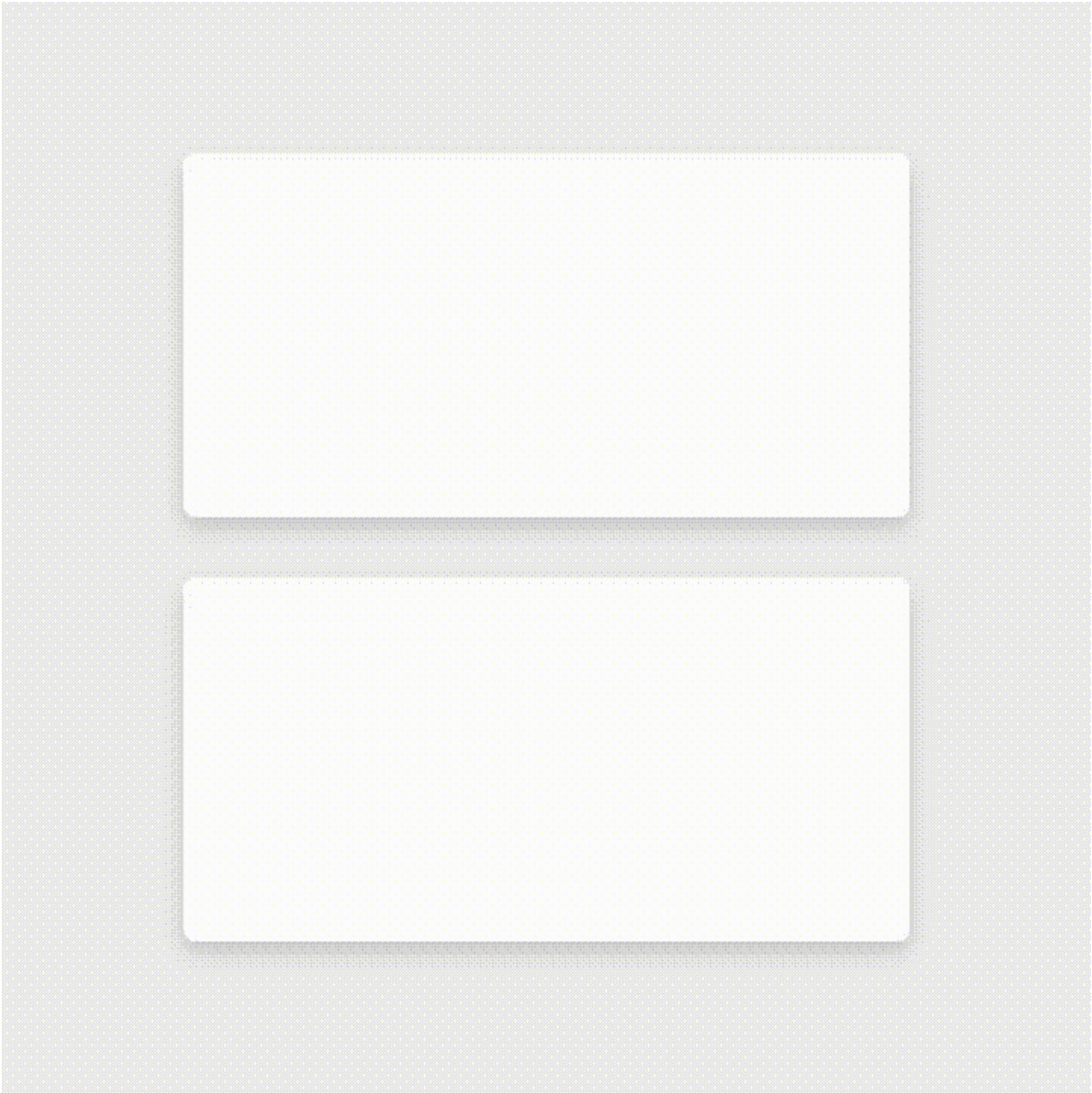
1. **Responsive**. It quickly responds to user input precisely where the user triggers it.
2. **Natural**. Movement is inspired by forces in the natural world. For example, real-world forces like gravity inspire an element's movement along an arc rather than in a straight line.



3. **Aware**. Material is aware of its surroundings, including the user and other material around it. Objects can be attracted to other objects in the UI, and they respond appropriately to user intent. As elements transition into view, their movement is choreographed in a way that defines their relationships.



4. **Intentional**. Movement guides the user's focus to the right place at the right time. Movement can communicate different signals, such as whether an action is unavailable.



To put these principles into practice in Android, use animations and transitions.

Animations

There are three ways you can create animation in your app:

- Property animation changes an object's properties over a specified period of time.The property animation system was introduced in Android 3.0 (API level 11). Property animation is more flexible than view animation, and it offers more features.
- View animation calculates animation using start points, end points, rotation, and other aspects of animation. The Android view animation system is older than the property animation system and can only be used for Views. It's relatively easy to set up and offers enough capabilities for many use cases.
- Drawable animation lets you load a series of drawable resources one after another to create an animation. Drawable animation is useful if you want to animate things that are easier to represent with drawable resources, such as a progression of bitmap images.

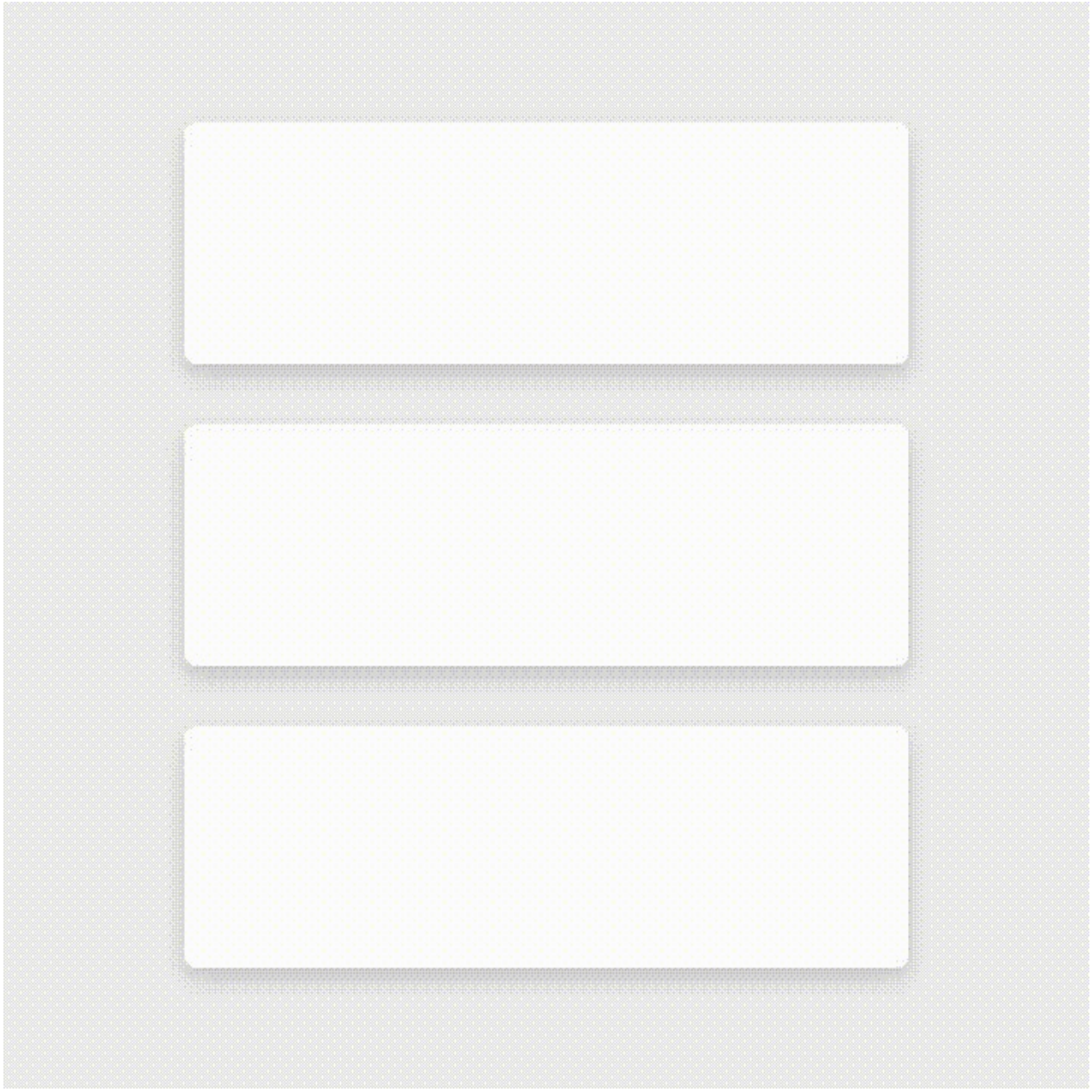
For complete details about these three types for animation, see the Animation and Graphics Overview.

The Material Design theme provides some default animations for touch feedback and activity transitions. The animation APIs let you create custom animations for touch feedback in UI controls, changes in view state, and activity transitions.

Touch feedback

Touch feedback provides instant visual confirmation at the point of contact when a user interacts with a UI element. The default touch feedback animations for buttons use the RippleDrawable class, which transitions between different states with a ripple effect.

In this example, ripples of ink expand outward from the point of touch to confirm user input. The card "lifts" and casts a shadow to indicate an active state:



In most cases, you apply ripple functionality in your view XML by specifying the view background as follows:

- `?android:attr/selectableItemBackground` for a bounded ripple.
- `?android:attr/selectableItemBackgroundBorderless` for a ripple that extends beyond the view. It is drawn upon, and bounded by, the nearest parent of the view with a non-null background.
  Note: The `selectableItemBackgroundBorderless` attribute was introduced in API level 21.

Alternatively, you can define a RippleDrawable as an XML resource using the `<ripple>` element.

You can assign a color to RippleDrawable objects. To change the default touch feedback color, use the theme's `android:colorControlHighlight` attribute.

For more information, see the API reference for the RippleDrawable class.

Circular reveal

A *reveal animation* shows or hides a group of UI elements by animating a view's clipping boundaries. In *circular reveal*, you reveal or hide a view by animating a clipping circle. (A *clipping circle* is a circle that crops or hides the part of an image that's outside the circle.)

To animate a clipping circle, use the `ViewAnimationUtils.createCircularReveal()` method. For example, here's how to reveal a previously invisible view using circular reveal:

```
// previously invisible view
View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the final radius for the clipping circle
float finalRadius = (float) Math.hypot(cx, cy);

// create the animator for this view (the start radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, 0, finalRadius);

// make the view visible and start the animation
myView.setVisibility(View.VISIBLE);
anim.start();
```

Here's how to hide a previously visible view using circular reveal:

```
// previously visible view
final View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the initial radius for the clipping circle
float initialRadius = (float) Math.hypot(cx, cy);

// create the animation (the final radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, initialRadius, 0);

// make the view invisible when the animation is done
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
        myView.setVisibility(View.INVISIBLE);
    }
});

// start the animation
anim.start();
```

Activity transitions

Activity transitions are animations that provide visual connections between different states in your UI. You can specify custom animations for *enter* and *exit* transitions, and for transitions of shared elements between activities.

- An *enter transition* determines how views in an activity enter the scene. For example in an *explode enter transition*, views enter the scene from the outside and fly towards the center of the screen.
- An *exit transition* determines how views in an activity exit the scene. For example in an *explode exit transition*, views exit the scene by moving away from the center.

- A *shared elements transition* determines how views that are shared between two activities transition between these activities. For example, if two activities have the same image in different positions and sizes, the `changeImageTransform` shared element transition translates and scales the image smoothly between these activities.

To use these transitions, set transition attributes in a `<style>` element in your XML. The following example creates a theme named `BaseAppTheme` that inherits one of the Material Design themes. The `BaseAppTheme` theme uses all three types of activity transitions:

```
<style name="BaseAppTheme" parent="android:Theme.Material">
  <!-- enable window content transitions -->
  <item name="android:windowActivityTransitions">true</item>

  <!-- specify enter and exit transitions -->
  <item name="android:windowEnterTransition">@transition/explode</item>
  <item name="android:windowExitTransition">@transition/explode</item>

  <!-- specify shared element transitions -->
  <item name="android:windowSharedElementEnterTransition">
    @transition/change_image_transform</item>
  <item name="android:windowSharedElementExitTransition">
    @transition/change_image_transform</item>
</style>
```

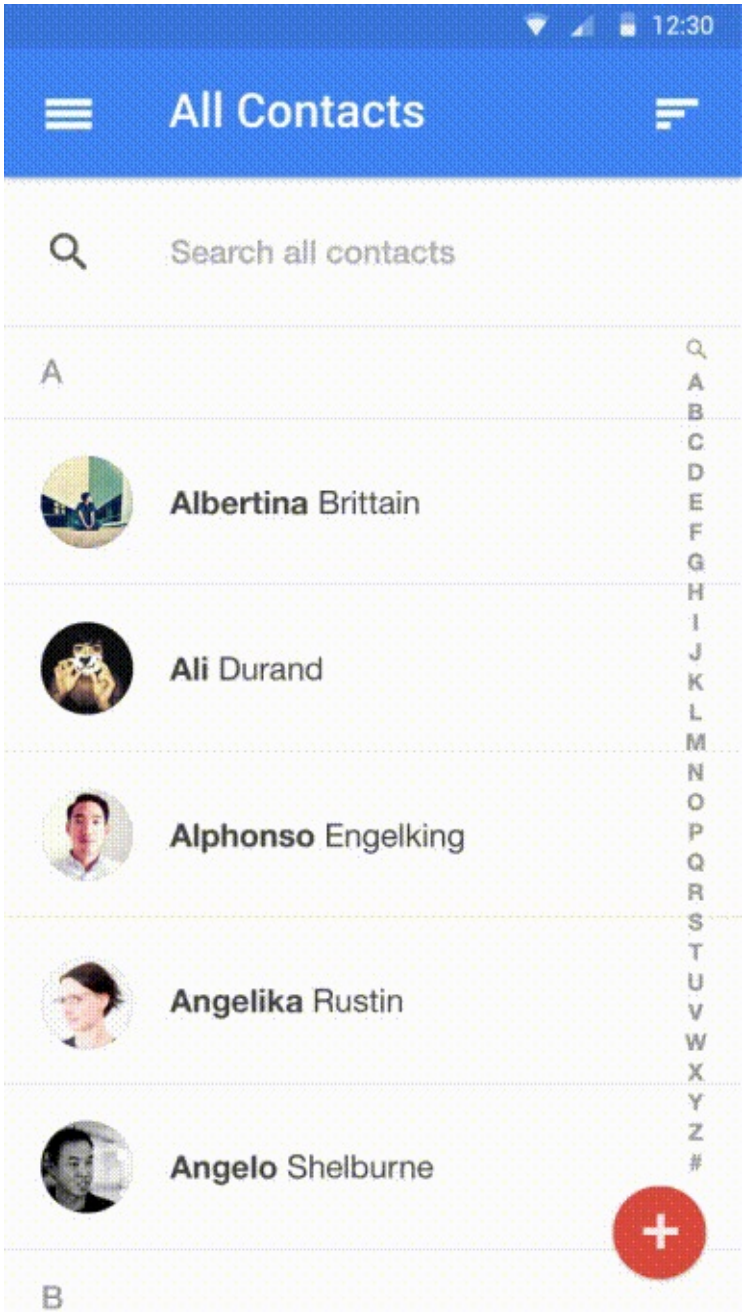
The `change_image_transform` transition in this example is defined as follows:

```
<!-- res/transition/change_image_transform.xml -->
<!-- (see also Shared Transitions below) -->
<transitionSet xmlns:android="http://schemas.android.com/apk/res/android">
  <changeImageTransform/>
</transitionSet>
```

The `changeImageTransform` element corresponds to the `ChangeImageTransform` class. For more information, see the API reference for `Transition`.

To enable window content transitions in your Java code instead, call the `Window.requestFeature()` method:

```
// inside your activity (if you did not enable transitions in your theme)
getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);

// set an exit transition
getWindow().setExitTransition(new Explode());
```


12:30
All Contacts
Search all contacts
A
Albertina Brittain
Ali Durand
Alphonso Engelking
Angelika Rustin
Angelo Shelburne
B
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z #

To specify transitions in your code, call the following methods with a `Transition` object:

- `Window.setEnterTransition()`
- `Window.setExitTransition()`
- `Window.setSharedElementEnterTransition()`
- `Window.setSharedElementExitTransition()`

For details about these methods, see the `Window reference documentation`.

To start an activity that uses transitions, use the `ActivityOptions.makeSceneTransitionAnimation()` method.

For more about implementing transitions in your app, see the activity transitions guide.

Curved motion

In Android 5.0 (API level 21) and above, you can define custom timing curves and curved motion patterns for animations. To do this, use the `PathInterpolator` class, which interpolates an object's path based on a Bézier curve or a `Path` object. The interpolator specifies a motion curve in a 1x1 square, with anchor points at (0,0) and (1,1) and control points that you specify using the constructor arguments. You can also define a path interpolator as an XML resource:

```
<pathInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:controlX1="0.4"
    android:controlY1="0"
    android:controlX2="1"
    android:controlY2="1"/>
```

The system provides XML resources for the three basic curves in the material design specification:

- `@interpolator/fast_out_linear_in.xml`
- `@interpolator/fast_out_slow_in.xml`
- `@interpolator/linear_out_slow_in.xml`

To use a `PathInterpolator` object, pass it to the `Animator.setInterpolator()` method.

The `ObjectAnimator` class has constructors you can use to animate coordinates along a path using two or more properties at once. For example, the following Java code uses a `Path` object to animate the X and Y properties of a view:

```
ObjectAnimator mAnimator;
mAnimator = ObjectAnimator.ofFloat(view, View.X, View.Y, path);
...
mAnimator.start();
```

Other custom animations

Other custom animations are possible, including animated state changes (using the `StateListAnimator` class) and animated vector drawables (using the `AnimatedVectorDrawable` class). For complete details, see Defining Custom Animations.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Material Design: Lists, Cards and Colors

Learn more

- Material Design for Android
- Material Design for Developers

5.3: Providing Resources for Adaptive Layouts

Table of Contents:

- Introduction
- Externalizing resources
- Grouping resources
- Alternative resources
- Creating alternative resources
- Common alternative-resource qualifiers
- Providing default resources
- Related practical
- Learn more

An *adaptive layo*ut is a layout that works well for different screen sizes and orientations, different devices, different locales and languages, and different versions of Android.

In this chapter you learn how to create an adaptive layout by externalizing and grouping resources, providing alternative resources, and providing default resources in your app.

Externalizing resources

When you *externalize* resources, you keep them separate from your application code. For example, instead of hard-coding a string into your code, you name the string and add it to the res/values/strings.xml file.

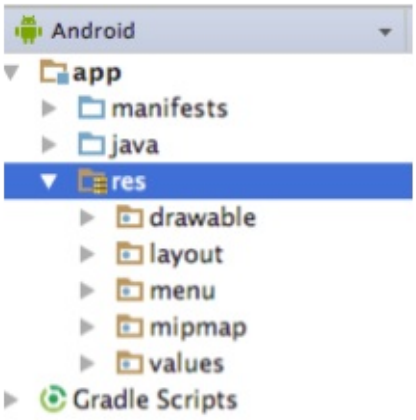
Always externalize resources such as drawables, icons, layouts, and strings. Here's why it's important:

- You can maintain externalized resources separately from your other code. If a resource is used in several places in your code and you need to change the resource, you only need to change it in one place.
- You can provide alternative resources that support specific device configurations, for example devices with different languages or screen sizes. This becomes increasingly important as more Android-powered devices become available.

Grouping resources

Store all your resources in the res/ folder. Organize resources by type into folders under /res. You must use standardized names for these folders.

For example, the screenshot below shows the file hierarchy for a small project, as seen in the "Android" Project view in Android Studio. The folders that contain this project's default resources use standardized names: `drawable`, `layout`,



`menu`, `mipmap` (for icons), and `values`.

Table 1 lists the standard resource folder names. The types are described more fully in the Providing Resources guide.

Table 1: Standard Resource Folder Names

Name	Resource Type
animator/	XML files that define property animations.
anim/	XML files that define tween animations.
color/	XML files that define "state lists" of colors. (This is different from the colors.xml file in the values/ folder.) See Color State List Resource.
drawable/	Bitmap files (WebP, PNG, 9-patch, JPG, GIF) and XML files that are compiled into drawables. See Drawable Resources.
mipmap/	Drawable files for different launcher icon densities. See Projects Overview.
layout/	XML files that define user interface layouts. See Layout Resource.
menu/	XML files that define application menus. See Menu Resource.
raw/	Arbitrary files saved in raw form. To open these resources with a raw InputStream, call Resources.openRawResource() with the resource ID, which is `R.raw.filename`. If you need access to original file names and file hierarchy, consider saving resources in the assets/ folder instead of res/raw/. Files in assets/ are not given a resource ID, so you can read them only using AssetManager.
values/	XML files that contain simple values, such as strings, integers, and colors. For clarity, place unique resource types in different files. For example, here are some filename conventions for resources you can create in this folder: • arrays.xml for resource arrays (typed arrays) • dimens.xml for dimension values • strings.xml, colors.xml, styles.xml See String Resources, Style Resource, and More Resource Types.
xml/	Arbitrary XML files that can be read at runtime by calling Resources.getXml(). Various XML configuration files, such as a searchable configuration, must be saved here, along with preference settings.

Alternative resources

Most apps provide *alternative resources* to support specific device configurations. For example, your app should include alternative drawable resources for different screen densities, and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources.

If no resources are available for the device's specific configuration, Android uses the default resources that you include in your app—the default drawables, which are in the res/drawable/ folder, the default text strings, which are in the res/values/strings.xml file, and so on.

Like default resources, alternative resources are kept in folders inside res/. Alternative-resource folders use the following naming convention:

```
<resource_name>-<config_qualifier>
```

- `<resource_name>` is the folder name for this type of resource, as shown in Table 1. For example, "drawable" or" values".
- `<config_qualifier>` specifies a device configuration for which these resources are used. The possible qualifiers are shown in Table 2.

  To add multiple qualifiers to one folder name, separate the qualifiers with a dash. If you use multiple qualifiers for a resource folder, you must list them in the order they are listed in Table 2.

Examples with one qualifier:

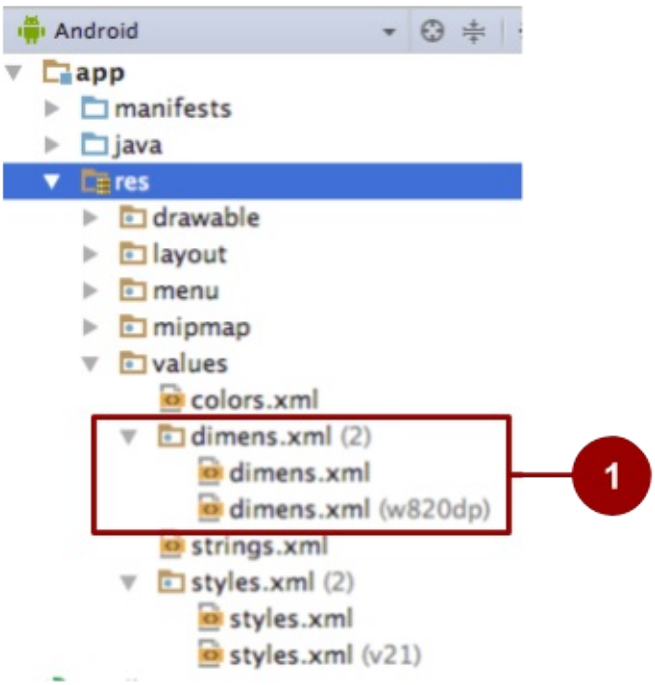
- String resources localized to Japanese would be in a `res/values-ja/strings.xml` file. Default string resources (resources to be used when no language-specific resources are found) would be in `res/values/strings.xml`. Notice that the XML files have identical names, in this case "strings.xml".
- Style resources for API level 21 and higher would be in a `res/values-v21/styles.xml` file. Default style resources would be in `res/values/styles.xml`.

Example with multiple qualifiers:

- Layout resources for a right-to-left layout running in "night" mode would be in a `res/layout-ldrtl-night/` folder.

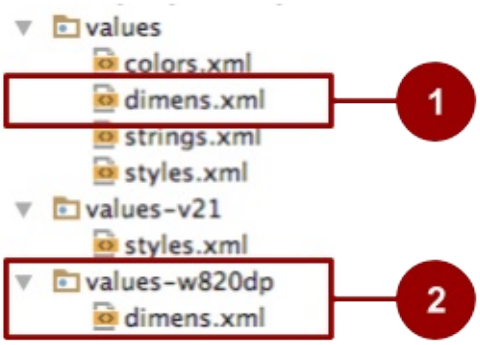
In the "Android" view in Android Studio, the qualifier is not appended to the end of the folder. Instead, the qualifier is shown as a label on the right side of the file in parentheses. For example, in the "Android" view shown below, the `res/values/dimens.xml/` folder shows two files:

- The `dimens.xml` file, which includes default dimension resources.
- The `dimens.xml (w820dp)` file, which includes dimension resources for devices that have a minimum available screen width of 820dp.



1. In the "Android" view in Android Studio, default resources for dimensions are shown in the same folder as alternative resources for dimensions.

In the "Project" view in Android Studio, the same information is presented differently, as shown in the screenshot below.



1. In the "Project" view in Android Studio, default resources for dimensions are shown in the `res/values` folder.
2. Alternative resources for dimensions are shown in `res/values-<qualifier>` folders.

Table 2 shows the configuration qualifiers that Android supports. They are listed in the order you must use when you combine multiple qualifiers in one folder name. For example in `res/layout-ldrtl-night/`, the qualifier for layout direction is listed before the qualifier for night mode, because layout direction is listed before night mode in the table.

These qualifiers are described in detail in Providing Alternative Resources.

Table 2: Qualifiers for Naming Alternative Resources

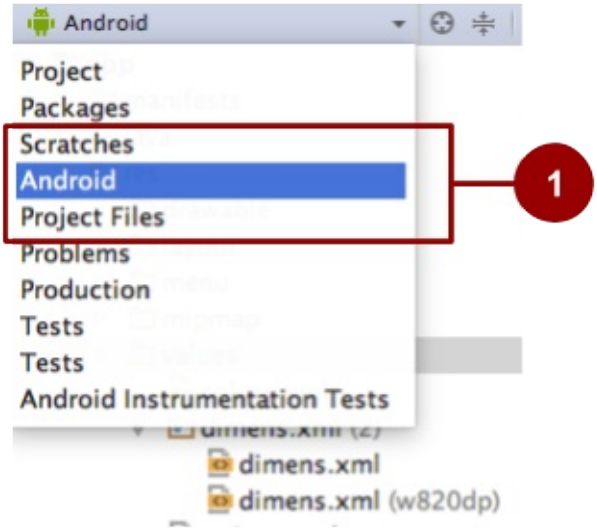
Precedence	Qualifier	Description
1	MCC and MNC	The mobile country code (MCC), optionally followed by mobile network code (MNC) from the SIM card in the device. For example, `mcc310` is U.S. on any carrier, `mcc310-mnc004` is U.S. on Verizon, and `mcc208-mnc00` is France on Orange.
2	Localization	Language, or language and region. Examples: `en`, `en-rUS`, `fr-rFR`, `fr-rCA`. Described in Localization, below.

3	Layout direction	The layout direction of your application. Possible values include `ldltr` (layout direction left-to-right, which is the default) and `ldrtl` (layout direction right-to-left). To enable right-to-left layout features, set supportsRtl to `"true"` and set targetSdkVersion to 17 or higher.
4	Smallest width	Fundamental screen size as indicated by the shortest dimension of the available screen area. Example: `sw320dp`. Described in Smallest width, below.
5	Available width	Minimum available screen width at which the resource should be used. Specified in dp units. The format is `wdp`, for example, `w720dp` and `w1024dp`.
6	Available height	Minimum available screen height at which the resource should be used. Specified in dp units. The format is `hdp`, for example, `h720dp` and `h1024dp`.
7	Screen size	Possible values: • `small:` Screens such as QVGA low-density screens • `normal:` Screens such as HVGA medium-density • `large:` Screens such as VGA medium-density • `xlarge:` Screens such as those on tablet-style devices
8	Screen aspect	Possible values include `long` (for screens such as WQVGA, WVGA, FWVGA) and `notlong` (for screens such as QVGA, HVGA, and VGA).
9	Round screen	Possible values include `round` (for screens such as those on round wearable devices) and `notround` (for rectangular screens such as phones).
10	Screen orientation	Possible values: `port, land.` Described in Screen orientation, below.
11	UI mode	Possible values: • `car:` Device is displaying in a car dock • `desk:` Displaying in a desk dock • `television:` Displaying on a large screen that the user is far away from, primarily oriented around D-pad or other non-pointer interaction • `appliance:` Device is serving as an appliance, with no display • `watch:` Device has a display and is worn on the wrist
12	Night mode	Possible values: • `night` • `notnight`
13	Screen pixel density	Possible values: • `ldpi:` Low-density screens; approximately 120dpi. • `mdpi:` Medium-density (on traditional HVGA) screens; approximately 160dpi. • `hdpi:` High-density screens; approximately 240dpi. • `xhdpi:` Approximately 320dpi. Added in API level 8. • `xxhdpi:` Approximately 480dpi. Added in API level 16. • `xxxhdpi:` Launcher icon only; approximately 640dpi. Added in API level 18. • `nodpi:` For bitmap resources that you don't want scaled to match the device density. • `tvdpi:` Screens between mdpi and hdpi; approximately 213dpi. Intended for televisions, and most apps shouldn't need it. Added in API level 13. • `anydpi:` Matches all screen densities and takes precedence over other qualifiers. Useful for vector drawables. Added in API level 21. **Note:** Using a density qualifier doesn't imply that the resources are *only* for screens of that density. If you don't provide alternative resources with qualifiers that better match the current device configuration, the system may use whichever resources are the best match.

14	Touchscreen type	Possible values include `notouch` (device doesn't have a touchscreen) and `finger` (device has a touchscreen).
15	Keyboard availability	Possible values: • `keysexposed:` Device has a keyboard available. • `keyshidden:` Device has a hardware keyboard available but it's hidden, and the device does not have a software keyboard enabled. • `keyssoft:` Device has a software keyboard enabled, whether it's visible or not.
16	Primary text input method	Possible values: • `nokeys:` Device has no hardware keys for text input. • `qwerty:` Device has a hardware qwerty keyboard, whether it's visible to the user or not. • `12key:` Device has a hardware 12-key keyboard, whether it's visible to the user or not.
17	Navigation key availability	Possible values include `navexposed` (navigation keys are available to the user) and `navhidden` (navigation keys are not available, for example they're behind a closed lid).
18	Primary non-touch navigation method	Possible values: • `nonav:` Device has no navigation facility other than the touchscreen. • `dpad:` Device has a directional-pad (D-pad). • `trackball:` Device has a trackball. • `wheel:` Device has a directional wheel for navigation (uncommon).
19	Platform version (API level)	The API level supported by the device. Described in Platform version, below.

Creating alternative resources

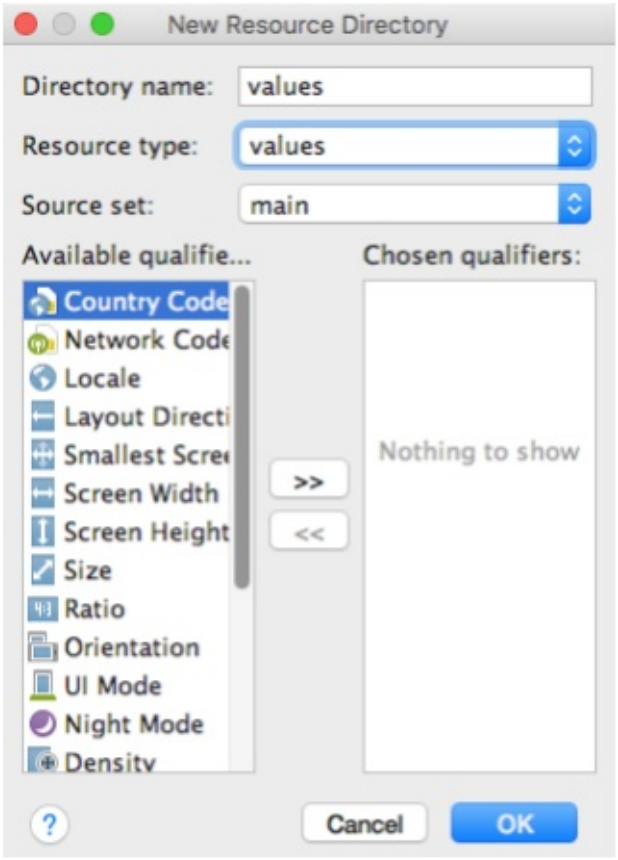
To create alternative resource folders most easily in Android Studio, use the "Android" view in the Project tool window.



1. Selecting the "Android" view in Android Studio. If you don't see these options, make sure the Project tool window is visible by selecting **View > Tool Windows > Project**.

To use Android Studio to create a new configuration-specific alternative resource folder in res/:

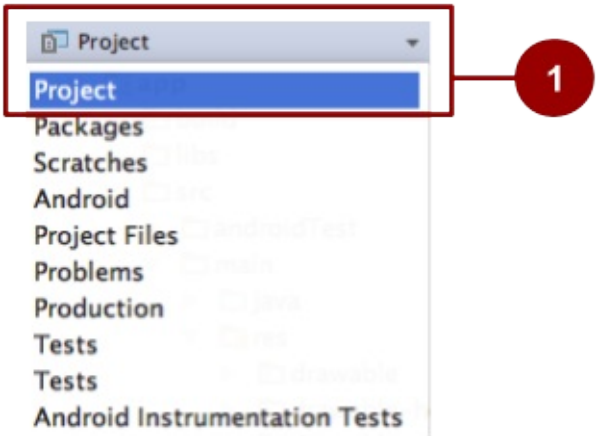
1. Be sure you are using the "Android" view, as shown above.
2. Right-click on the res/ folder and select **New > Android resource directory**. The New Resource Directory dialog box



appears.

3. Select the type of resource (described in Table 1) and the qualifiers (described in Table 2) that apply to this set of alternative resources.
4. Click OK.

If you can't see the new folder in the Project tool window in Android Studio, switch to the "Project" view, as shown in the screenshot below. If you don't see these options, make sure the Project tool window is visible by selecting **View > Tool**



Windows > Project.

Save alternative resources in the new folder. The alternative resource files must be named exactly the same as the default resource files, for example "styles.xml" or "dimens.xml".

For the complete documentation about alternative resources, see Providing Alternative Resources.

Common alternative-resource qualifiers

This section describes a few commonly used qualifiers. Table 2 gives the complete list.

Screen orientation

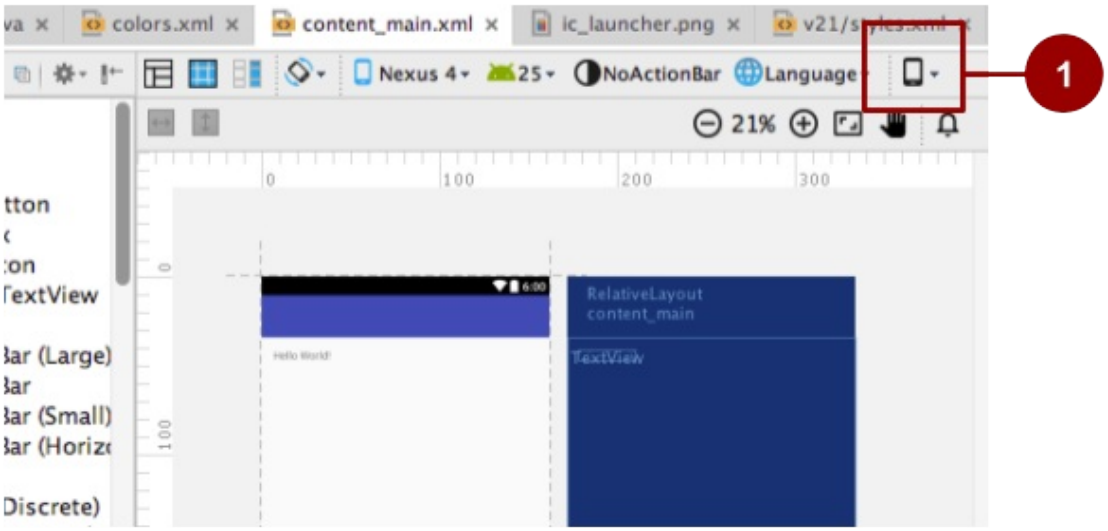
The screen-orientation qualifier has two possible values:

- `port` : The device is in portrait mode (vertical). For example, `res/layout-port/` would contain layout files to use when the device is in portrait mode.
- `land` : The device is in landscape mode (horizontal). For example, `res/layout-land/` would contain layout files to use when the device is in landscape mode.

If the user rotates the screen while your app is running, and if alternative resources are available, Android automatically reloads your app with alternative resources that match the new device configuration. For information about controlling how your app behaves during a configuration change, see Handling Runtime Changes.

To create variants of your layout XML file for landscape orientation and larger displays, use the layout editor. To use the layout editor:

1. In Android Studio, open the XML file. The layout editor appears.
2. From the drop-down menu in the **Layout Variants** menu, choose an option such as **Create Landscape Variant**. The **Layout Variants** menu, which is visible when an XML file is open in Android Studio, is highlighted in the screenshot below.



A layout for a different landscape orientation appears, and a new XML file is created for you. For example, you might now have a file named "activity_main.xml (land)" along with your original "activity_main.xml" file. You can use the editor to change the new layout without changing the original layout.

See the previous practical about layouts for an example of layout design.

Smallest width

The smallest-width qualifier specifies the minimum width of the device. It is the shortest of the screen's available height and width, the "smallest possible width" for the screen. The smallest width is a fixed screen-size characteristic of the device, and it does not change when the screen's orientation changes.

Specify smallest width in dp units, using the following format:

```
sw<N>dp
```

where `<N>` is the minimum width. For example, resources in a file named `res/values-sw320dp/styles.xml` are used if the device's screen is always at least 320dp wide.

You can use this qualifier to ensure that a certain layout won't be used unless it has at least `<N>` dps of width available to it, regardless of the screen's current orientation.

Some values for common screen sizes:

- 320, for devices with screen configurations such as
  - 240x320 ldpi (QVGA handset)
  - 320x480 mdpi (handset)
  - 480x800 hdpi (high-density handset)
- 480, for screens such as 480x800 mdpi (tablet/handset)
- 600, for screens such as 600x1024 mdpi (7" tablet)
- 720, for screens such as 720x1280 mdpi (10" tablet)

When your application provides multiple resource folders with different values for the smallest-width qualifier, the system uses the one closest to (without exceeding) the device's smallest width.

Example:

`res/values-sw600dp/dimens.xml` contains dimensions for images. When the app runs on a device with a smallest width of 600dp or higher (such as a tablet), Android uses the images in this folder.

Platform version

The platform-version qualifier specifies the minimum API level supported by the device. For example, use `v11` for API level 11 (devices with Android 3.0 or higher). See the Android API levels document for more information about these values.

Use the platform-version qualifier when you use resources for functionality that's unavailable in prior versions of Android.

For example, WebP images require API level 14 (Android 4.0) or higher, and for full support they require API level 17 (Android 4.2) or higher. If you use WebP images:

- Put default versions of the images in a `res/drawable` folder. These images must use an image format that's supported for all API levels, for example PNG.
- Put WebP versions of the images in a `res/drawable-v17` folder. If the device uses API level 17 or greater, Android will select these resources at runtime.

Localization

The localization qualifier specifies a language and, optionally, a region. This qualifier is a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase `r`).

You can specify a language alone, but not a region alone. Examples:

- `res/values-fr-rFR/strings.xml`

  Strings in this file are used on devices that are configured for the French language and have their region set to France.

- `res/mipmap-fr-rCA/`

  Icons in this folder are used on devices that are configured for the French language and have their region set to Canada.

- `res/layout-ja/content_main.xml`

  This layout is used on devices that are configured for the Japanese language.

If the user changes the language or region in the device's system settings while your app is running, and if alternative resources are available, Android automatically reloads your app with alternative resources that match the new device configuration. For information about controlling how your app behaves during a configuration change, see Handling Runtime Changes.

For a full guide on localization, see Localizing with Resources.

Providing default resources

Default resources specify the default design and content for your application. For example, when the app runs in a locale for which you have not provided locale-specific text, Android loads the default strings from res/values/strings.xml. If this default file is absent, or if it is missing even one string that your application needs, then your app doesn't run and shows an error.

Default resources have standard resource folder names (`values` , for example) without any qualifiers in the folder name or

in parentheses after the file names.

1. Default resources
   Tip: Always provide default resources, because your app might run on a device configuration that you don't anticipate.

Sometimes new versions of Android add configuration qualifiers that older versions don't support. If you use a new resource qualifier and maintain code compatibility with older versions of Android, then when an older version of Android runs your app, the app crashes unless default resources are available. This is because the older version of Android can't use the alternative resources that are named with the new qualifier.

For example, assume your `minSdkVersion` is set to `4` and you qualify all your drawable resources using night mode, meaning that you put all your drawable resources in `res/drawable-night/` and `res/drawable-notnight/` . In this example:

- When an API level 4 device runs the app, the device can't access your drawable resources. The Android version doesn't know about `night` and `notnight` , because these qualifiers weren't added until API level 8. The app crashes, because it doesn't include any default resources to fall back on.

In this example, you probably want `notnight` to be your default case. To solve the problem, exclude the `notnight` qualifier and put your drawable resources in `res/drawable/` and `res/drawable-night/` . With this solution:

- When an API level 4 device runs the app, it always uses the resources in the default `res/drawable/` folder.
- When a device at API level 8 or above uses the app, it uses the resources in the `res/drawable-night/` folder whenever the device is in night mode. At all other times, it uses the default (`notnight`) resources.

To provide the best device compatibility, provide default resources for every resource that your application needs. After your default resources are in place, create alternative resources for specific device configurations using the alternative-resource configuration qualifiers shown in Table 2.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Supporting Landscape, Multiple Screen Sizes and Localization

Learn more

- Providing Resources
- Resources Overview

- Localizing with Resources

6.1: Testing the User Interface

Contents:

- Properly testing a user interface
- Setting up your test environment
- Using Espresso for tests that span a single app
- Using UI Automator for tests that span multiple apps
- Related practical
- Learn more

Properly testing a user interface

Writing and running tests are important parts of the Android app development cycle. Well-written tests can help you catch bugs early in your development cycle, where they are easier to fix, and helps give you confidence in your code.

User interface (UI) testing focuses on testing aspects of the user interface and interactions with users. Recognizing and acting on user input is a high priority in user interface testing and validation. You need to make sure that your app not only recognizes the type of input but also acts accordingly. As a developer, you should get into the habit of testing user interactions to ensure that users don't encounter unexpected results or have a poor experience when interacting with your app. User interface testing can help you recognize the input controls where unexpected input should be handled gracefully or should trigger input validation.

Note: We strongly encourage you to use Android Studio for building your test apps, because it provides project setup, library inclusion, and packaging conveniences. You can run UI tests on a variety of physical or virtual Android devices, and you can then analyze the results and make changes to your code without leaving the development environment.
The UI contains views with graphical elements such as buttons, menus, and text fields, each with a set of properties. To properly test the UI, you need to:

- Exercise all UI events with views:
  - Tap a UI view, and enter data or make a choice.
  - Examine the *values of the properties* of each view—referred to as the *state* of the UI—at different times during execution.
- Provide inputs to all UI views. Use this opportunity to test improper inputs, such as text when numbers are expected.
- Check the outputs and UI representations of data—such as strings and integers—to see if they are consistent with what you are expecting.

In addition to functionality, UI testing evaluates design elements such as layout, colors, fonts, font sizes, labels, text boxes, text formatting, captions, buttons, lists, icons, links, and content.

Manual Testing

As the developer of an app, you probably test each UI component manually as you add the component to the app's UI. As development continues, one approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly.

However, this manual approach can be time-consuming, tedious, and error-prone. By manually testing a UI for a complex app, you can't possibly cover all permutations of user interactions. You would also have to manually perform these repetitive tests on many different device configurations in an emulator, and on many different devices. To summarize, the problems inherent with manual testing fall into two categories:

- *Domain size:* A UI has a great deal of operations that need testing. Even a relatively small app can have hundreds of possible UI operations. Over a development cycle a UI may change significantly, even though the underlying app doesn't change. Manual tests with instructions to follow a certain path through the UI may fail over time, because a

button, menu item, or dialog may have changed location or appearance.
- *Sequences:* Some functionality of the app may only be accomplished with a sequence of UI events. For example, to add an image to a message about to be sent, a user may have to tap a camera button and use the camera to take a picture, or a photo button to select an existing picture, and then associate that picture with the message—usually by tapping a share or send button. Increasing the number of possible operations also increases the sequencing problem.

Automated testing

When you automate tests of user interactions, you free yourself and your resources for other work. To generate a set of test cases, test designers attempt to cover all of the functionality of the system and fully exercise the UI. Performing all of the UI interactions automatically makes it easier to run tests for different device states (such as orientations) and different configurations.

For testing Android apps, you typically create these types of automated UI tests:

- *UI tests that work within a single app:* Verifies that the app behaves as expected when a user performs a specific action or enters a specific input. It allows you to check that the app returns the correct UI output in response to user interactions in the app's activities. UI testing frameworks like Espresso allow you to programmatically simulate user actions and test complex intra-app user interactions.
- *UI tests that span multiple apps:* Verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, you can test an app that launches the Maps app to show directions, or that launches the Android contact picker to select recipients for a message. UI testing frameworks that support cross-app interactions, such as UI Automator, allow you to create tests for such user-driven scenarios.

Using Espresso for tests that span a single app

The Espresso testing framework in the Android Testing Support Library provides APIs for writing UI tests to simulate user interactions within a single app. Espresso tests run on actual device or emulator and behave as if an actual user is using the app.

You can use Espresso to create UI tests to automatically verify the following:

- The app returns the correct UI output in response to a sequence of user actions on a device.
- The app's navigation and input controls bring up the correct activities, views, and fields.
- The app responds correctly with mocked-up dependencies, such as data from an outside server, or can work with stubbed out backend methods to simulate real interactions with backend components which can be programmed to reply with a set of defined responses.

A key benefit of using Espresso is that it has access to instrumentation information, such as the context for the app, so that you can monitor all of the interaction the Android system has with the app. Another key benefit is that it automatically synchronizes test actions with the app's UI. Espresso detects when the main thread is idle, so it is able to run your test at the appropriate time, improving the reliability of your tests. This capability also relieves you from having to add any timing workarounds, such as a sleep period, in your test code.

The Espresso testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described later in this section. Espresso tests can run on devices running Android 2.2 (API level 8) and higher.

Using UI Automator for tests that span multiple apps

The UI Automator testing framework in the Android Testing Support Library can help you verify the correct behavior of interactions between different user apps or between user apps and system apps. It can also show you what is happening on the device before and after an app is launched.

The UI Automator APIs let you interact with visible elements on a device. Your test can look up a UI component by using descriptors such as the text displayed in that component or its content description. A viewer tool provides a visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device.

The following are important functions of UI Automator:

- Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.
- Your test can send an Intent or launch an Activity (without using shell commands) by getting a Context object through getContext().
- You can simulate user interactions on a collection of items, such as songs in a music album or a list of emails in an inbox.
- You can simulate vertical or horizontal scrolling across a display.
- You can use standard JUnit Assert methods to test that UI components in the app return the expected results.

The UI Automator testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described in the next section. UI Automator tests can run on devices running Android 4.3 (API level 18) or higher.

What is instrumentation?

Android instrumentation is a set of control methods, or *hooks*, in the Android system, which control Android components and how the Android system loads apps.

Normally the system runs all the components of an app in the same process. You can allow some components, such as content providers, to run in a separate process, but you typically can't force an app onto the same process as another running app.

Instrumentation tests, however, can load both a test package and the app into the same process. Since the app components and their tests are in the same process, your tests can invoke methods in the components, and modify and examine fields in the components.

Instrumentation allows you to monitor all of the interaction the Android system has with the application, and makes it possible for tests to invoke methods in the app, and modify and examine fields in the app, independently of the app's normal lifecycle.

Normally, an Android component runs in a lifecycle that the system determines. For example, an Activity object's lifecycle starts when an Intent activates the Activity. The system calls the object's onCreate() method, and then the onResume() method. When the user starts another app, the system calls the onPause() method. If the Activity code calls the finish() method, the system calls the onDestroy() method.

The Android framework API does not provide a way for your app's code to invoke these callback methods directly, but you can do so using an Espresso or UI Automator test with instrumentation.

Setting up your test environment

To use the Espresso and UI Automator frameworks, you need to store the source files for instrumented tests at ***module-name*/src/androidTests/java/**. This directory already exists when you create a new Android Studio project. In the Project view of Android Studio, this directory is shown in **app > java** as ***module-name*** **(androidTest)**.

You also need to do the following:

- Install the Android Support Repository and the Android Testing Support Library.
- Add dependencies to the project's build.gradle file.
- Create test files in the **androidTest** directory.

Installing the Android Support Repository and Testing Support Library

You may already have the Android Support Repository and its Android Testing Support Library installed with Android Studio. To check for the Android Support Repository, follow these steps:

1. In Android Studio choose **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab, and look for the Support Repository.

3. If necessary, update or install the library.

Adding the dependencies

When you start a project for the Phone and Tablet form factor using **API 15: Android 4.0.3 (Ice Cream Sandwich)** as the minimum SDK, Android Studio version 2.2 and newer automatically includes the dependencies you need to use Espresso. To ensure that you have these dependencies, follow these steps:

1. Open the **build.gradle (Module: app)** file in your project to make sure the following is included (along with other dependencies) in the `dependencies` section of your **build.gradle (Module: app)** file:

```
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
testCompile 'junit:junit:4.12'
```

2. Android Studio also adds the following instrumentation statement to the end of the `defaultConfig` section:

```
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
```

Note: If you created your project in a previous version of Android Studio, you may have to add the above dependencies and instrumentation statement yourself.

1. When finished, click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

Setting up the test rules and annotations

To write tests, Espresso and UI Automator use JUnit as their testing framework. JUnit is the most popular and widely-used unit testing framework for Java. Your test class using Espresso or UI Automator should be written as a JUnit 4 test class. If you do not have JUnit, you can get it at http://junit.org/junit4/.

Note: The most current JUnit revision is JUnit 5. However for the purposes of using Espresso or UI Automator, version 4.12 is recommended.
To create a basic JUnit 4 test class, create a Java class for testing in the directory specified at the beginning of this section. It should contain one or more methods and behavior rules defined by JUnit annotations.

For example, the following snippet shows a test class definition with annotations:

```
@RunWith(AndroidJUnit4.class)
public class RecyclerViewTest {

   @Rule
   public ActivityTestRule<MainActivity> mActivityTestRule =
            new ActivityTestRule<>(MainActivity.class);

   @Test
   public void recyclerViewTest() {
      ...
   }
}
```

The following annotations are useful for testing:

@RunWith

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition, which indicates the runner that will be used to run the tests in this class. A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log.

@SmallTest, @MediumTest, and @LargeTest

The `@SmallTest`, `@MediumTest`, and `@LargeTest` Android annotations provide some clarity about what resources and features the test uses. For example, the `@SmallTest` annotation tells you that the test doesn't interact with any file system or network.

The following summarizes what they mean:

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

For a description of the Android `@SmallTest`, `@MediumTest`, and `@LargeTest` annotations, see "Test Sizes" in the Google Testing Blog. For a summary of JUnit annotations, see Package org.junit.

@Rule

Before declaring the test methods, use the `@Rule` annotation, such as ActivityTestRule or ServiceTestRule. The `@Rule` establishes the context for the testing code. For example:

```
@Rule
public ActivityTestRule mActivityRule = new ActivityTestRule<>(
                    MainActivity.class);
```

This rule uses an `ActivityTestRule` object, which provides functional testing of a single Activity—in this case, `MainActivity.class`.

`ServiceTestRule` is a JUnit rule that provides a simplified mechanism to start and shutdown your service before and after the duration of your test. It also guarantees that the service is successfully connected when starting (or binding to) a service.

@Test

A test method begins with the `@Test` annotation and contains the code to exercise and verify a single function in the component that you want to test:

```
@Test
public void testActivityLaunch() { … }
```

The activity under test will be launched before each test annotated with `@Test`. During the duration of the test you will be able to manipulate your Activity directly.

@Before and @After

In certain rare cases you may need to set up variables or execute a sequence of steps *before* or *after* performing the user interface test. You can specify methods to run before running a `@Test` method, using the `@Before` annotation, and methods to run after, using the `@After` annotation.

- `@Before` : The `@Test` method will execute after the methods designated by the `@Before` annotation. The `@Before` methods terminate prior to the execution of the `@Test` method.
- `@After` : The `@Test` method will execute before the methods designated by the `@After` annotation.

Using Espresso for tests that span a single app

When writing tests it is sometimes difficult to get the balance right between over-specifying the test or not specifying enough. Over-specifying the test can make it brittle to changes, while under-specifying may make the test less valuable, since it continues to pass even when the UI element and its code under test is broken.

To make tests that are balanced, it helps to have a tool that allows you to pick out precisely the aspect under test and describe the values it should have. Such tests fail when the behavior of the aspect under test deviates from the expected behavior, yet continue to pass when minor, unrelated changes to the behavior are made. The tool available for Espresso is the Hamcrest framework.

Hamcrest (an anagram of "matchers") is a framework that assists writing software tests in Java. The framework lets you create custom assertion matchers, allowing match rules to be defined declaratively.

Tip: To learn more about Hamcrest, see The Hamcrest Tutorial.

Writing Espresso tests with Hamcrest Matchers

You write Espresso tests based on what a user might do while interacting with your app. The key concepts are *locating* and then *interacting* with UI elements. These are the basic steps:

1. **Match a view:** Find a view.
2. **Perform an action:** Perform a click or other action that triggers an event with the view.
3. **Assert and verify the result:** Check the view's state to see if it reflects the expected state or behavior defined by the assertion.

To create a test method, you use the following types of Hamcrest expressions to help find views and interact with them:

- *ViewMatchers:* A ViewMatcher expression to use with `onView()` to match a view in the current view hierarchy. Use `onView()` with a ViewMatcher so that you can examine something or perform some action. The most common ones are:
  - `withId()` : Find a view with a specific `android:id` (which is typically defined in the layout XML file). For example:

    ```
    onView(withId(R.id.my_view))
    ```

  - `withText()` : Find a view with specific text, typically used with `allOf()` and `withId()` . For example, the following uses `allOf` to cause a match if the examined object matches *all* of the specified conditions—the view uses the `word` id (`withId`), and the view has the text `"Clicked! Word 15"` (`withText`):

    ```
    onView(allOf(withId(R.id.word),
         withText("Clicked! Word 15"), isDisplayed()))
    ```

  - Others including matchers for state (`selected` , `focused` , `enabled`), and content description and hierarchy (`root` and `children`).
- *ViewActions:* A ViewAction expression lets you perform an action on the view already found by a ViewMatcher. The action can be any action you can perform on the view, such as a click. For example:

  ```
  .perform(click())
  ```

- *ViewAssertions:* A ViewAssertion expression lets you assert or checks the state of a view found by a ViewMatcher. For example:

```
.check(matches(isDisplayed()))
```

You would typically combine a ViewMatcher and a ViewAction in a single statement, followed by a ViewAssertion expression in a separate statement or included in the same statement.

You can see how all three expressions work in the following statement, which combines a ViewMatcher to find a view, a ViewAction to perform an action, and a ViewAssertion to check if the result of the action matches an assertion:

```
onView(withId(R.id.my_view))           // withId(R.id.my_view) is a ViewMatcher
        .perform(click())              // click() is a ViewAction
        .check(matches(isDisplayed())); // matches(isDisplayed()) is a ViewAssertion
```

Why is the Hamcrest framework useful for tests? A simple assertion, such as `assert (x == y)`, lets you assert during a test that a particular condition must be true. If the condition is false, the test fails. But the simple assertion provides no useful error message. With a family of assertions, you can produce more useful error messages, but this leads to an explosion in the number of assertions.

With the Hamcrest framework, it is possible to define operations that take matchers as arguments and return them as results, leading to a grammar that can generate a huge number of possible matcher expressions from a small number of primitive matchers.

For a Hamcrest tutorial, see The Hamcrest Tutorial. For a quick summary of Hamcrest matcher expressions, see the Espresso cheat sheet.

Testing an AdapterView

In an AdapterView such as a spinner, the view is dynamically populated with child views at runtime. If the target view you want to test is inside a spinner, the `onView()` method might not work because only a subset of the views may be loaded in the current view hierarchy.

Espresso handles this by providing a separate `onData()` method, which is able to first load the adapter item and bring it into focus prior to operating on it or any of its children views. The `onData()` method uses a DataInteraction object and its methods, such as `atPosition()`, `check()`, and `perform()` to access the target view. Espresso handles loading the target view element into the current view hierarchy, scrolling to the target child view, and putting that view into focus.

For example, the following `onView()` and `onData()` statements test a spinner item click:

1. Find and click the spinner itself (the test must first click the spinner itself in order click any item in the spinner):

```
onView(withId(R.id.spinner_simple)).perform(click());
```

2. Find and then click the item in the spinner that matches *all* of the following conditions:
   - An item that is a `String`
   - An item that is equal to the String "Americano"

```
onData(allOf(is(instanceOf(String.class)),
                        is("Americano"))).perform(click());
```

As you can see in the above statement, matcher expressions can be combined to create flexible expressions of intent:

- `allOf`: Causes a match if the examined object matches *all* of the specified matchers. You can use `allOf()` to combine multiple matchers, such as `containsString()` and `instanceOf()`.
- `is`: Hamcrest strives to make your tests as readable as possible. The `is` matcher is a wrapper that doesn't add any extra behavior to the underlying matcher, but makes your test code more readable.
- `instanceOf`: Causes a match when the examined object is an instance of the specified type; in this case, a string. This match is determined by calling the Class.isInstance(Object) method, passing the object to examine.

The following example illustrates how you would test a spinner using a combination of `onView()` and `onData()` methods:

```
@RunWith(AndroidJUnit4.class)
public class SpinnerSelectionTest {

   @Rule
   public ActivityTestRule mActivityRule = new ActivityTestRule<>(
            MainActivity.class);

   @Test
   public void iterateSpinnerItems() {
      String[] myArray = mActivityRule.getActivity().getResources()
                              .getStringArray(R.array.labels_array);

      // Iterate through the spinner array of items.
      int size = myArray.length;
      for (int i=0; i<size; i++) {
         // Find the spinner and click on it.
         onView(withId(R.id.label_spinner)).perform(click());
         // Find the spinner item and click on it.
         onData(is(myArray[i])).perform(click());
         // Find the button and click on it.
         onView(withId(R.id.button_main)).perform(click());
         // Find the text view and check that the spinner item
         // is part of the string.
         onView(withId(R.id.text_phonelabel))
                    .check(matches(withText(containsString(myArray[i]))));
      }
   }
}
```

The test clicks each spinner item from top to bottom, checking to see if the item appears in the text field. It doesn't matter how many spinner items are defined in the array, or what language is used for the spinner's items—the test performs all of them and checks their output against the array.

The following is a step-by-step description of the above test:

1. The `iterateSpinnerItems()` method begins by getting the array used for the spinner items:

   ```
   public void iterateSpinnerItems() {
   String[] myArray =
        mActivityRule.getActivity().getResources()
        .getStringArray(R.array.labels_array);
   ...
   ```

   In the statement above, the test accesses the application's array (with the id `labels_array`) by establishing the context with the `getActivity()` method of the ActivityTestRule class, and getting a resources instance in the application's package using `getResources()`.

2. Using the length (`size`) of the array, the `for` loop iterates through each spinner item.

   ```
   ...
   int size = myArray.length;
   for (int i=0; i<size; i++) {
   // Find the spinner and click on it.
   ...
   ```

3. The `onView()` statement within the `for` loop finds the spinner and clicks on it. The test must click the spinner itself in order click any item in the spinner:

   ```
   ...
   // Find the spinner and click on it.
   onView(withId(R.id.label_spinner)).perform(click());
   ...
   ```

4. The `onData()` statement finds and clicks a spinner item:

   ```
   ...
   // Find the spinner item and click on it.
   onData(is(myArray[i])).perform(click());
   ...
   ```

   The spinner is populated from the `myArray` array, so `myArray[i]` represents a spinner element from the array. As the `for` loop iterates `for (int i=0; i<size; i++)`, it performs a click on each spinner element (`myArray[i])` it finds.

5. The last `onView()` statement finds the text view (`text_phonelabel`) and checks that the spinner item is part of the string:

   ```
   ...
   onView(withId(R.id.text_phonelabel))
        .check(matches(withText(containsString(myArray[i]))));
   ...
   ```

Using RecyclerViewActions for a Recycler View

A RecyclerView is useful when you have data collections with elements that change at runtime based on user action or network events. RecyclerView is a UI component designed to render a collection of data, but is not a subclass of AdapterView but of ViewGroup. This means that you can't use `onData()`, which is specific to AdapterView, to interact with list items.

However, there is a class called RecyclerViewActions that exposes a small API to operate on a RecyclerView. For example, the following test clicks on an item from the list by position:

```
onView(withId(R.id.recyclerView))
      .perform(RecyclerViewActions.actionOnItemAtPosition(0, click()));
```

The following test class demonstrates how to use RecyclerViewActions to test a RecyclerView. The app lets you scroll a list of words. When you click on a word such as **Word 15** the word in the list changes to "Clicked! Word 15":

```
@RunWith(AndroidJUnit4.class)
public class RecyclerViewTest {

   @Rule
   public ActivityTestRule<MainActivity> mActivityTestRule =
            new ActivityTestRule<>(MainActivity.class);

   @Test
   public void recyclerViewTest() {
      ViewInteraction recyclerView = onView(
            allOf(withId(R.id.recyclerview), isDisplayed()));
      recyclerView.perform(actionOnItemAtPosition(15, click()));

      ViewInteraction textView = onView(
            allOf(withId(R.id.word), withText("Clicked! Word 15"),
                 childAtPosition(
                         childAtPosition(
                                 withId(R.id.recyclerview),
                                 11),
                         0),
                 isDisplayed()));
      textView.check(matches(withText("Clicked! Word 15")));

   }

   private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {

      return new TypeSafeMatcher<View>() {
         @Override
         public void describeTo(Description description) {
            description.appendText("Child at position "
                                        + position + " in parent ");
            parentMatcher.describeTo(description);
         }

         @Override
         public boolean matchesSafely(View view) {
            ViewParent parent = view.getParent();
            return parent instanceof ViewGroup &&
                                    parentMatcher.matches(parent)
                                    && view.equals(((ViewGroup)
                                    parent).getChildAt(position));
         }
      };
   }
}
```

The test uses a recyclerView object of the ViewInteraction class, which is the primary interface for performing actions or assertions on views, providing both `check()` and `perform()` methods. Each interaction is associated with a view identified by a view matcher:

- The code below uses the `perform()` method and the actionOnItemAtPosition() method of the RecyclerViewActions class to scroll to the position (15) and click the item:

  ```
  ViewInteraction recyclerView = onView(
        allOf(withId(R.id.recyclerview), isDisplayed()));
  recyclerView.perform(actionOnItemAtPosition(15, click()));
  ```

- The code below checks to see if the clicked item matches the assertion that it should be `"Clicked! Word 15"`:

```
    ViewInteraction textView = onView(
          allOf(withId(R.id.word), withText("Clicked! Word 15"),
               childAtPosition(
                       childAtPosition(
                               withId(R.id.recyclerview),
                               11),
                       0),
               isDisplayed()));
    textView.check(matches(withText("Clicked! Word 15")));
```

- The code above uses a method called `childAtPosition()`, which is defined as a custom `Matcher`:

```
private static Matcher<View> childAtPosition(
      final Matcher<View> parentMatcher, final int position) {
   // TypeSafeMatcher() returned
   ...
   }
```

The custom matcher extends the abstract TypeSaveMatcher class and requires that you implement the following:

- The `matchesSafely()` method to define how to check for a view in a RecyclerView.
- The `describeTo()` method to define how Espresso describes the output's matcher in the Run pane at the bottom of Android Studio if a failure occurs.

```
    ...
    // TypeSafeMatcher() returned
    return new TypeSafeMatcher<View>() {
       @Override
       public void describeTo(Description description) {
          description.appendText("Child at position "
                                      + position + " in parent ");
          parentMatcher.describeTo(description);
       }

       @Override
       public boolean matchesSafely(View view) {
          ViewParent parent = view.getParent();
          return parent instanceof ViewGroup &&
                                  parentMatcher.matches(parent)
                                  && view.equals(((ViewGroup)
                                  parent).getChildAt(position));
       }
    };
  }
}
```

Recording a test

An Android Studio feature (in version 2.2 and newer) lets you *record* an Espresso test, creating the test automatically.

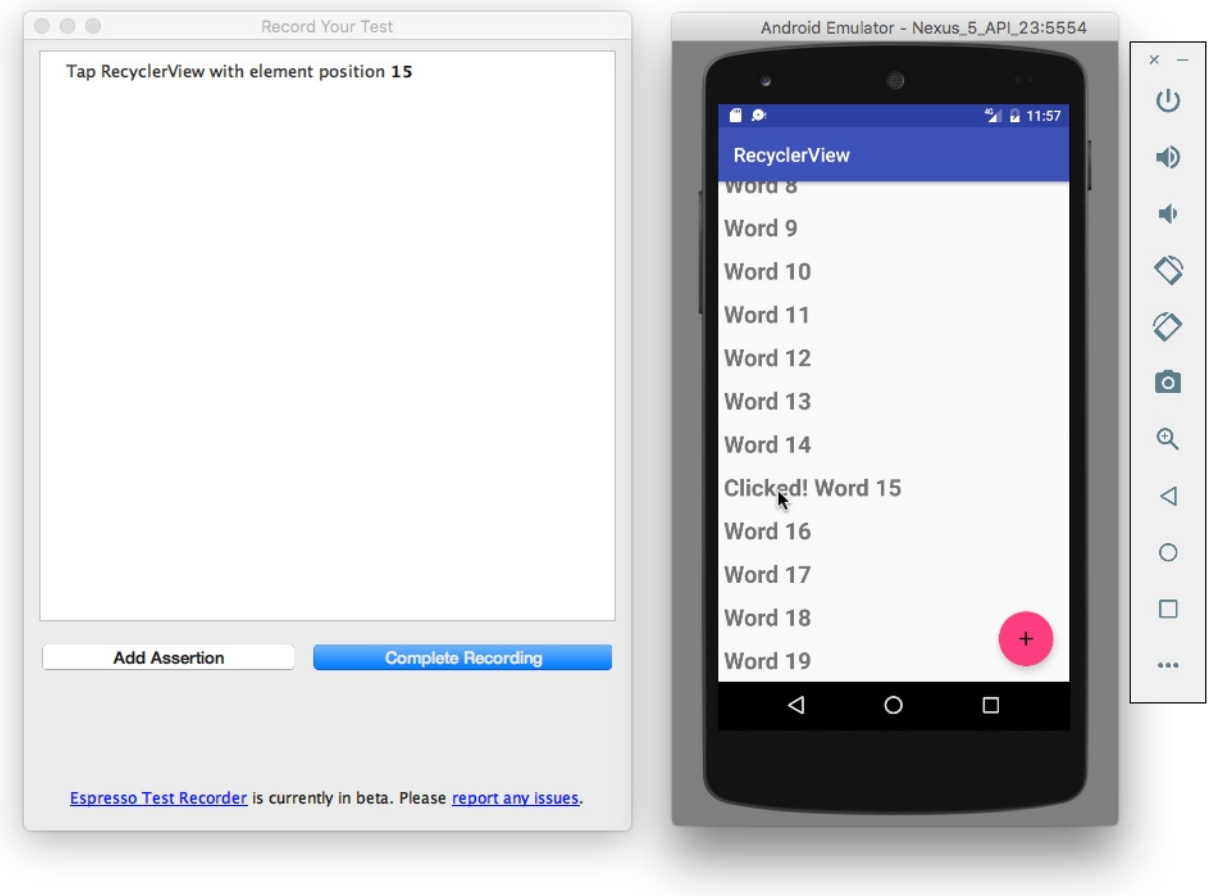
After choosing to record a test, use your app as a normal user would. As you click through the app UI, editable test code is generated for you. Add assertions to check if a view holds a certain value.

You can record multiple interactions with the UI in one recording session. You can also record multiple tests, and edit the tests to perform more actions, using the recorded code as a snippet to copy, paste, and edit.

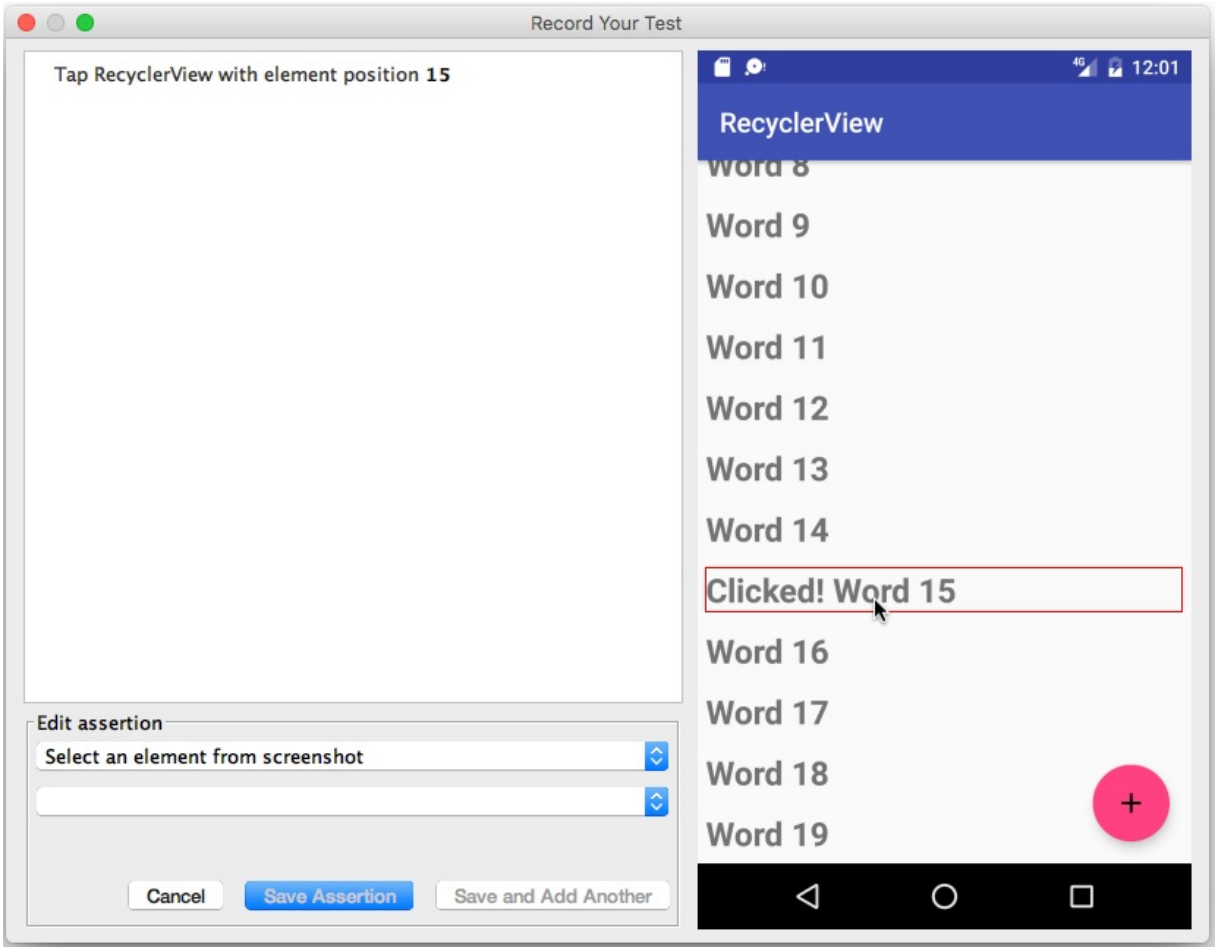
Follow these steps to record a test, using the RecyclerView app as an example:

Android Studio Project: RecyclerView

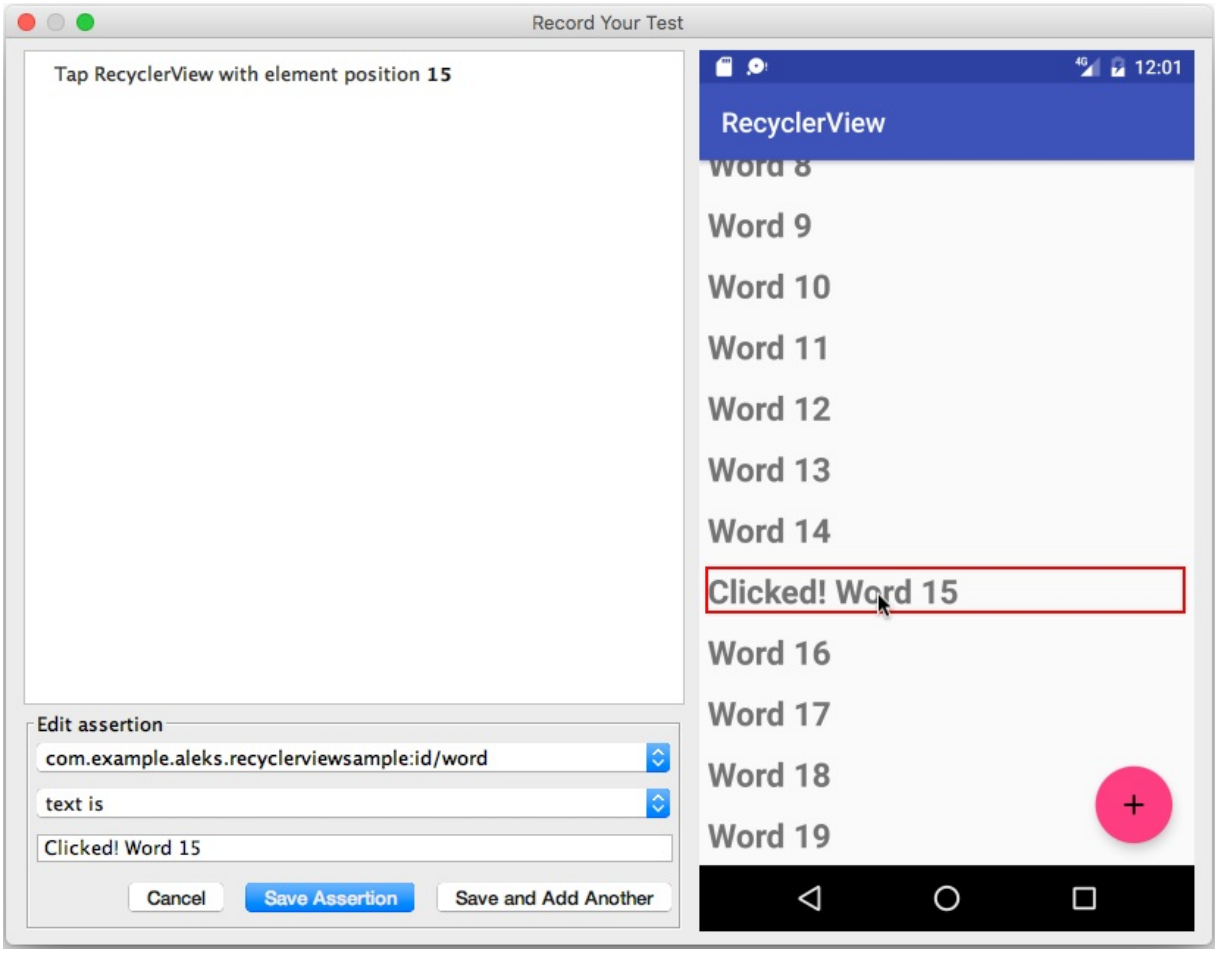
1. Choose **Run > Record Espresso Test**, select your deployment target (an emulator or a device), and click **OK**.
2. Interact with the UI to do what you want to test. In this case, scroll the word list in the app on the emulator or a device, and tap **Word 15**. The Record Your Test window shows the action that was recorded ("Tap RecyclerView with element position 15").



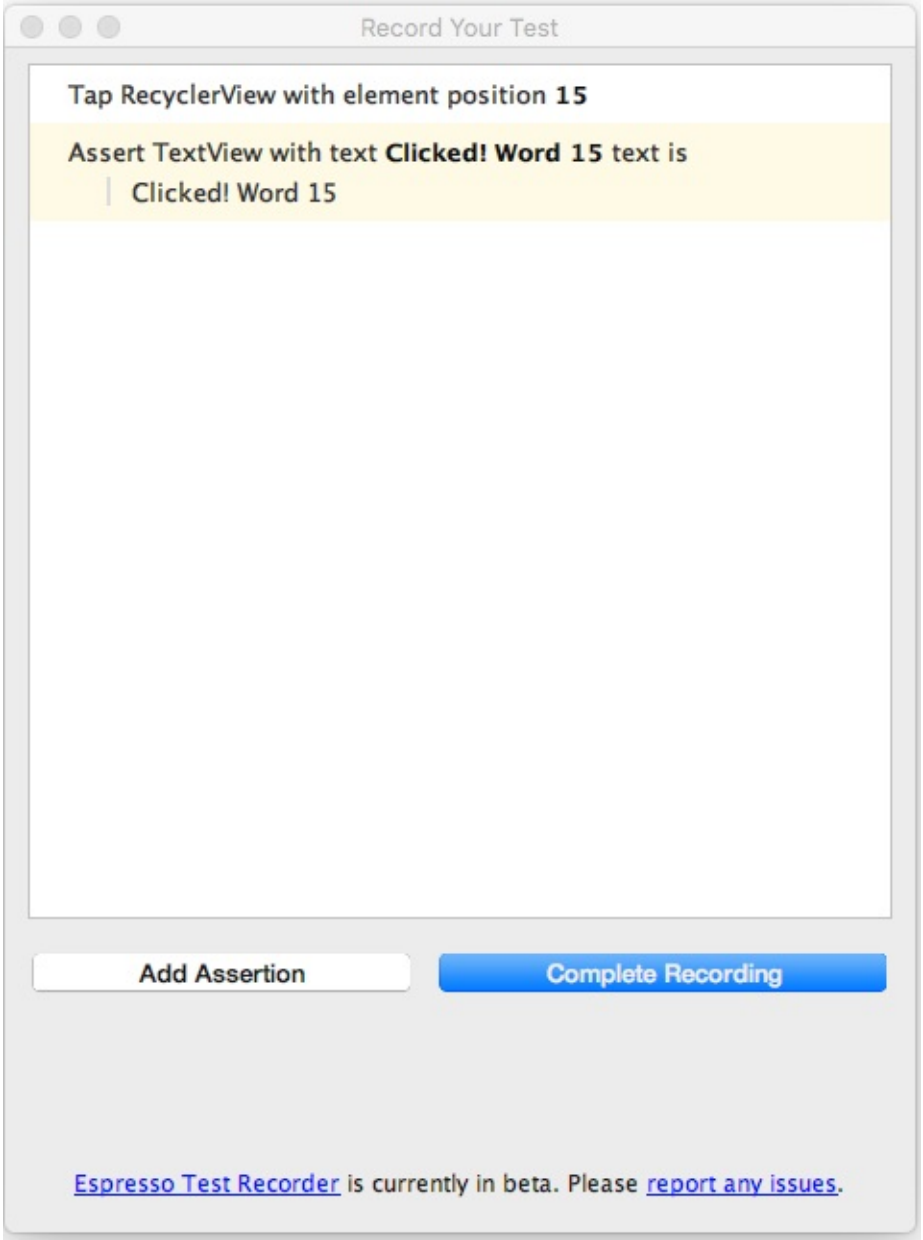
3. Click **Add Assertion** in the Record Your Test window. A screenshot of the app's UI appears in a pane on the right side of the window. Select **Clicked! Word 15** in the screenshot as the UI element you want to check. This creates an assertion for the selected element's view.



4. Choose **text is** from the second drop-down menu, and enter the text you expect to see in that UI element.



5. Click **Save Assertion**, and then click **Complete Recording**.


Record Your Test
Tap RecyclerView with element position 15
Assert TextView with text Clicked! Word 15 text is
Clicked! Word 15
Add Assertion
Complete Recording
Espresso Test Recorder is currently in beta. Please report any issues.

6. In the dialog that appears, you can edit the name of the test, or accept the name suggested (such as **MainActivityTest**).
7. Android Studio may display a request to add more dependencies to your Gradle Build file. Click **Yes** to add the dependencies.

Using UI Automator for tests that span multiple apps

UI Automator is a set of APIs that can help you verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. A viewer tool provides a visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.

To use UI Automator, you must already have set up your test environment in the same way as for Espresso:

- Install the Android Support Repository and the Android Testing Support Library.
- Add the following dependency to the project's build.gradle file:

```
androidTestCompile
     'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
```

Use UI Automator Viewer to Inspect the UI on a device

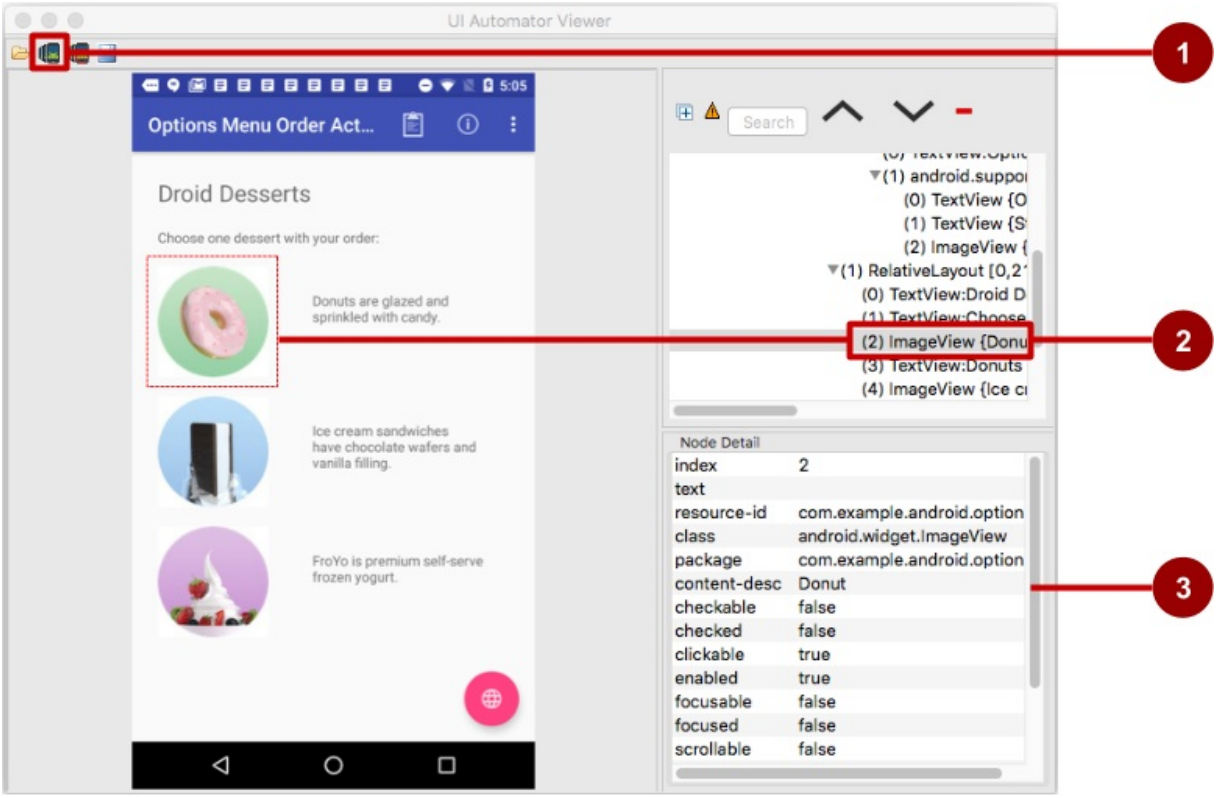
UI Automator Viewer (uiautomatorviewer) provides a convenient visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device.

To launch the uiautomatorviewer tool, follow these steps:

1. Install and then launch the app on a physical device such as a smartphone.
2. Connect the device to your development computer.
3. Open a terminal window and navigate to the **/tools/** directory. To find the specific path, choose **Preferences** in Android Studio, and click **Appearance & Behavior > System Settings > Android SDK**. The full path for appears in the Android SDK Location box at the top of the screen.
4. Run the tool with this command: **uiautomatorviewer**

To ensure that your UI Automator tests can access the app's UI elements, check that the elements have visible text labels, android:contentDescription values, or both. You can view the properties of a UI element by following these steps (refer to the figure below):

1. After launching uiautomatorviewer, the viewer is empty. Click the **Device Screenshot** button.
2. Hover over a UI element in the snapshot in the left-hand panel to see the element in the layout hierarchy in the upper right panel.
3. The layout attributes and other properties for the UI element appear in the lower right panel. Use this information to create tests that select a UI element that matches a specific visible attribute or property.

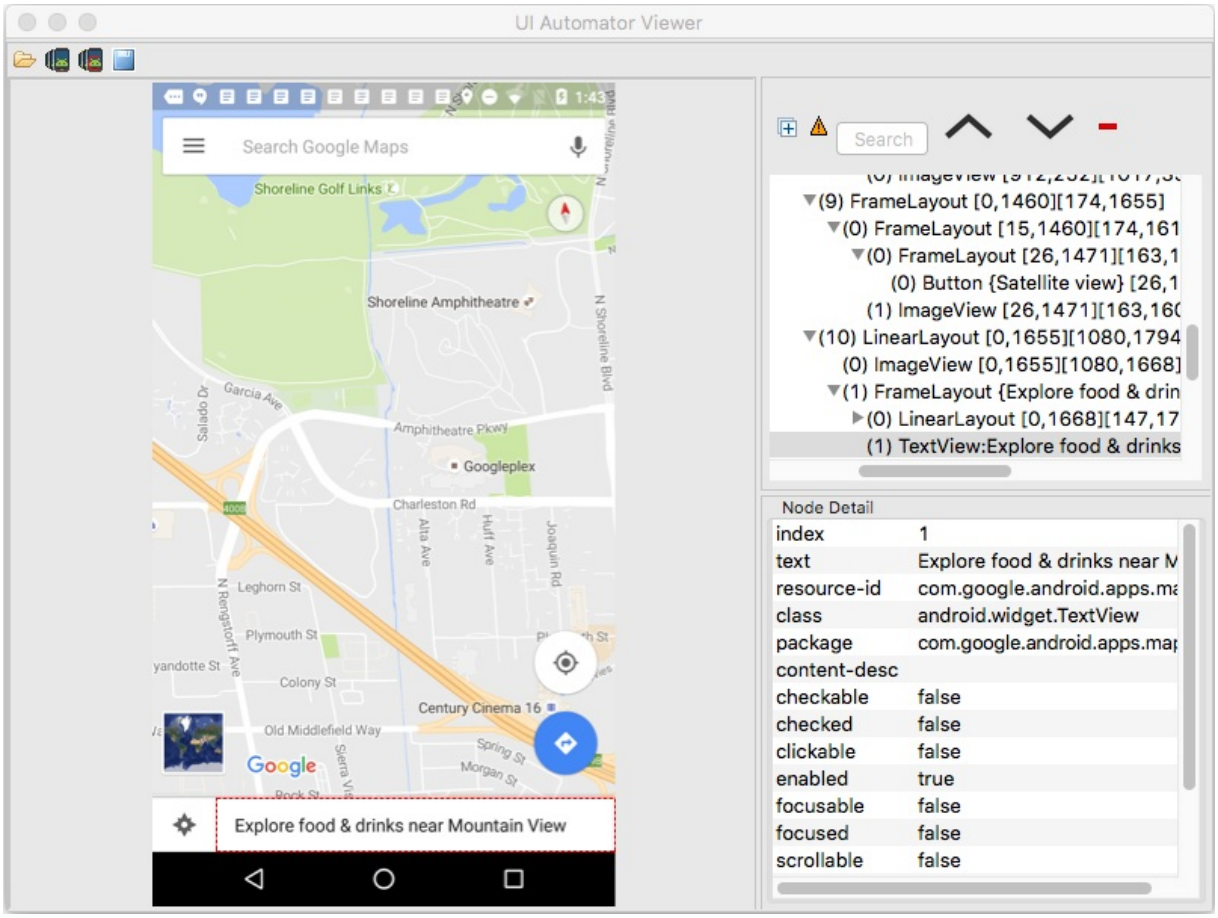


In the above figure:

1. **Device Screenshot** button
2. Selected component in the snapshot and in the layout hierarchy
3. Properties for the selected component

In the app shown above, the red floating action button launches the Maps app. Follow these steps to test the action performed by the floating action button:

1. Tap the floating action button.
2. The code for the button makes an implicit intent to launch the Maps app.
3. Click the **Device Screenshot** button to see the result of the implicit intent (the Maps screen).



Using the viewer, you can determine which UI elements are accessible to the UI Automator framework.

Setup: Ensuring that UI elements are accessible

The UI Automator framework depends on the accessibility features of the Android framework to look up individual UI elements. Implement view properties as follows:

- Include the android:contentDescription attribute in your XML layout to label the ImageButton, ImageView, CheckBox, and other user input controls. The following shows the `android:contentDescription` attribute added to a `RadioButton` using the same string resource tused for the `android:text` attribute:

```
<RadioButton
  android:id="@+id/sameday"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/same_day_messenger_service"
  android:contentDescription="@string/same_day_messenger_service"
  android:onClick="onRadioButtonClicked"/>
```

  Tip: You can make input controls more accessible for the sight-impaired by using the `android:contentDescription` XML layout attribute. The text in this attribute does not appear on screen, but if the user enables accessibility services that provide audible prompts, then when the user navigates to that control, the text is spoken.

- Provide an android:hint attribute for EditText fields (in addition to `android:contentDescription`, which is useful for

accessibility services). With EditText fields, UI Automator looks for the `android:hint` attribute.
- Associate an `android:hint` attribute with any graphical icons used by controls that provide feedback to the user (for example, status or state information).

As a developer, you should implement the above minimum optimizations to support your users as well as UI Automator.

Creating a test class

A UI Automator test class generally follows this programming model:

1. **Access the device to test:** An instance of the InstrumentRegistry class holds a reference to the instrumentation running in the process along with the instrumentation arguments. It also provides an easy way for callers to get instrumentation, application context, and an instrumentation arguments Bundle. You can get a UiDevice object by calling the getInstance() method and passing it an Instrumentation object — `InstrumentationRegistry.getInstrumentation()` —as the argument. For example:

   ```
   mDevice = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());
   ```

2. **Access a UI element displayed on the device:** Get a UiObject by calling the findObject() method. For example:

   ```
   UiObject okButton = mDevice.findObject(new UiSelector()
       .text("OK"))
       .className("android.widget.Button"));
   ```

3. **Perform an action:** Simulate a specific user interaction to perform on that UI element by calling a UiObject method. For example:

   ```
   if(okButton.exists() && okButton.isEnabled()) {
       okButton.click();
   }
   ```

   You may want to call setText() to edit a text field, or performMultiPointerGesture() to simulate a multi-touch gesture.

   You can repeat steps 2 and 3 as needed to test more complex user interactions that involve multiple UI components or sequences of user actions.

4. **Verify results:** Check that the UI reflects the expected state or behavior after these user interactions are performed. You can use standard JUnit Assert methods to test that UI components in the app return the expected results. For example:

   ```
   UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
   assertEquals("5", result.getText());
   ```

Accessing the device

The UiDevice class provides the methods for accessing and manipulating the state of the device. Unlike Espresso, UI Automator can verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. In your tests, you can call UiDevice methods to check for the state of various properties, such as current orientation or display size. Your test can use the UiDevice object to perform device-level actions, such as forcing the device into a specific rotation, pressing D-pad hardware buttons, and pressing the Home button.

It's a good practice to start your test from the Home screen of the device. From the Home screen you can call the methods provided by the UI Automator API to select and interact with specific UI elements. The following code snippet shows how your test can get an instance of UiDevice, simulate a Home button press, and launch the app:

```
import org.junit.Before;
import android.support.test.runner.AndroidJUnit4;
import android.support.test.uiautomator.UiDevice;
import android.support.test.uiautomator.By;
import android.support.test.uiautomator.Until;
...

@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {

    private static final String BASIC_SAMPLE_PACKAGE
            = "com.example.android.testing.uiautomator.BasicSample";
    private static final int LAUNCH_TIMEOUT = 5000;
    private static final String STRING_TO_BE_TYPED = "UiAutomator";
    private UiDevice mDevice;

    @Before
    public void startMainActivityFromHomeScreen() {
        // Initialize UiDevice instance
        mDevice =
          UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());

        // Start from the home screen
        mDevice.pressHome();

        // Wait for launcher
        final String launcherPackage = mDevice.getLauncherPackageName();
        assertThat(launcherPackage, notNullValue());
        mDevice.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
                LAUNCH_TIMEOUT);

        // Launch the app
        Context context = InstrumentationRegistry.getContext();
        final Intent intent = context.getPackageManager()
                .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
        // Clear out any previous instances
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(intent);

        // Wait for the app to appear
        mDevice.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
                LAUNCH_TIMEOUT);
    }
}
```

The `@SdkSuppress(minSdkVersion = 18)` annotation ensures that tests will only run on devices with Android 4.3 (API level 18) or higher, as required by the UI Automator framework.

Accessing a UI element

Use the findObject() method of the UiObject class to retrieve a UiObject instance that represents a UI element matching a given selector criteria. To access a specific UI element, use the UiSelector class, which represents a query for specific elements in the currently displayed UI.

You can reuse the UiObject instances that you created in other parts of your app testing. The UI Automator test framework searches the current display for a match every time your test uses a UiObject instance to click on a UI element or query an attribute.

The following shows how your test might construct UiObject instances using `findObject()` with a UiSelector that represent a **Cancel** button, and one that represents an **OK** button:

```
UiObject cancelButton = mDevice.findObject(new UiSelector()
        .text("Cancel"))
        .className("android.widget.Button"));
UiObject okButton = mDevice.findObject(new UiSelector()
        .text("OK"))
        .className("android.widget.Button"));
```

If more that one element matches, the first matching element in the layout hierarchy (found by moving from top to bottom, left to right) is returned as the target UiObject. When constructing a UiSelector, you can chain together multiple attributes and properties to refine your search. If no matching UI element is found, an exception (UiAutomatorObjectNotFoundException) is thrown.

To nest multiple UiSelector instances, use the childSelector() method of the UiSelector class. For example, the following shows how your test might specify a search to find the first ListView in the currently displayed UI, then search within that ListView to find a UI component with the `android:text` attribute `"List Item 14"`:

```
UiObject appItem = new UiObject(new UiSelector()
        .className("android.widget.ListView")
        .instance(1)
        .childSelector(new UiSelector()
        .text("List Item 14")));
```

While it may be useful to refer to the `android:text` attribute of an element of a ListView or RecycleView because there is no resource id (`android:id` attribute) for such an element, it is best to use a resource id when specifying a selector rather than the `android:text` or `android:contentDescription` attributes. Not all elements have a text attribute (for example, icons in a toolbar). Tests might fail if there are minor changes to the text of a UI component, and the tests would not be usable for apps translated into other languages because your text selectors would not match the translated string resources.

Performing actions

Once your test has retrieved a UiObject object, you can call the methods in the UiObject class to perform user interactions on the UI component represented by that object. For example, the constructed UiObject instances in the previous section for the OK and Cancel buttons can be used to perform a click:

```
// Simulate a user-click on the OK button, if found.
if(okButton.exists() && okButton.isEnabled()) {
    okButton.click();
}
```

You can use UiObject methods to perform actions such as:

- click(): Click the center of the visible bounds of the UI element.
- dragTo(): Drag the object to arbitrary coordinates.
- setText(): Set the text in an editable field, after clearing the field's content. Conversely, you use the clearTextField() method to clear the existing text in an editable field.
- swipeUp(): Perform the swipe up action on the UiObject. Similarly, the swipeDown(), swipeLeft(), and swipeRight() methods perform corresponding actions.

Sending an intent or launching an activity

The UI Automator testing framework enables you to send an Intent or launch an Activity without using shell commands, by getting a Context object through the getContext() method. For example, the following shows how your test can use an Intent to launch the app under test:

```
public void setUp() {
    ...
    // Launch a simple calculator app
    Context context = getInstrumentation().getContext();
    Intent intent = context.getPackageManager()
            .getLaunchIntentForPackage(CALC_PACKAGE);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
            // Clear out any previous instances
    context.startActivity(intent);
    mDevice.wait(Until.hasObject(By.pkg(CALC_PACKAGE).depth(0)), TIMEOUT);
}
```

Performing actions on collections

Use the UiCollection class if you want to simulate user interactions on a collection of UI elements (for example, song titles or emails in a list). To create a UiCollection object, specify a UiSelector that searches for a UI container or a wrapper of other child UI elements, such as a layout group that contains child UI elements.

The following shows how a test can use a UiCollection to represent a video album that is displayed within a FrameLayout:

```
UiCollection videos = new UiCollection(new UiSelector()
        .className("android.widget.FrameLayout"));

// Retrieve the number of videos in this collection:
int count = videos.getChildCount(new UiSelector()
        .className("android.widget.LinearLayout"));

// Find a specific video and simulate a user-click on it
UiObject video = videos.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "Cute Baby Laughing");
video.click();

// Simulate selecting a checkbox that is associated with the video
UiObject checkBox = video.getChild(new UiSelector()
        .className("android.widget.Checkbox"));
if(!checkBox.isSelected()) checkbox.click();
```

Performing actions on scrollable views

Use the UiScrollable class to simulate vertical or horizontal scrolling across a display. This technique is helpful when a UI element is positioned off-screen and you need to scroll to bring it into view. For example, the following code snippet shows how to simulate scrolling down the **Settings** menu and clicking on the **About phone** option:

```
UiScrollable settingsItem = new UiScrollable(new UiSelector()
        .className("android.widget.ListView"));
UiObject about = settingsItem.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "About phone");
about.click();
```

Verifying results

You can use standard JUnit Assert methods to test that UI components in the app return the expected results. For example, you can use assertFalse() to assert that a condition is false in order to test if the condition truly is false as a result. Use assertEquals() to test if a floating point number result is equal to the assertion:

```
assertEquals("5", result.getText());
```

The following shows how your test can locate several buttons in a calculator app, click on them in order, then verify that the correct result is displayed:

```
private static final String CALC_PACKAGE = "com.myexample.calc";
public void testTwoPlusThreeEqualsFive() {
    // Enter an equation: 2 + 3 = ?
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("two")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("plus")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("three")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("equals")).click();

    // Verify the result = 5
    UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
    assertEquals("5", result.getText());
}
```

Running instrumented tests

To run a single test, right-click (or Control-click) the test in Android Studio, and choose **Run** from the pop-up menu.

To test a method in a test class, right-click the method in the test file and click **Run**.

To run all tests in a directory, right-click on the directory and select **Run tests**.

Android Studio displays the results of the test in the Run window.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Use Espresso to Test Your UI

Learn more

Android Studio Documentation:

- Test Your App

Android Developer Documentation:

- Best Practices for Testing
- Getting Started with Testing
- Testing UI for a Single App—Espresso
- Testing UI for Multiple Apps—UI Automator
- Building Instrumented Unit Tests
- Espresso Advanced Samples
- The Hamcrest Tutorial
- Hamcrest API and Utility Classes
- Test Support APIs

Android Testing Support Library:

- Espresso documentation
- Espresso Samples
- Espresso basics
- Espresso cheat sheet

Videos

- Android Testing Support - Android Testing Patterns #1 (introduction)
- Android Testing Support - Android Testing Patterns #2 (onView view matching)
- Android Testing Support - Android Testing Patterns #3 (onData and adapter views)

Other:

- Google Testing Blog:
  - Android UI Automated Testing
  - Test Sizes
- Atomic Object: "Espresso – Testing RecyclerViews at Specific Positions"
- Stack Overflow: "How to assert inside a RecyclerView in Espresso?"
- GitHub: Android Testing Samples
- Google Codelabs: Android Testing Codelab
- JUnit web site
- JUnit annotations: Package org.junit

7.1: AsyncTask and AsyncTaskLoader

Contents:

- The UI thread
- AsyncTask
- AsyncTask usage
- Example of an AsyncTask
- Executing an AsyncTask
- Cancelling an AsyncTask
- Limitations of AsyncTask
- Loaders
- AsyncTaskLoader
- AsyncTaskLoader usage
- Related practical
- Learn more

There are two ways to do background processing in Android: using the `AsyncTask` class, or using the `Loader` framework, which includes an `AsyncTaskLoader` class that uses `AsyncTask`. In most situations you'll choose the `Loader` framework, but it's important to know how `AsyncTask` works so you can make a good choice.

In this chapter you'll learn why it's important to process some tasks in the background, off the UI thread. You'll learn how to use `AsyncTask`, when *not* to use `AsyncTask`, and the basics of using loaders.

The UI thread

When an Android app starts, it creates the *main thread*, which is often called the *UI thread*. The UI thread dispatches events to the appropriate user interface (UI) widgets, and it's where your app interacts with components from the Android UI toolkit (components from the android.widget and android.view packages).

Android's thread model has two rules:

1. Do not block the UI thread.

The UI thread needs to give its attention to drawing the UI and keeping the app responsive to user input. If everything happened on the UI thread, long operations such as network access or database queries could block the whole UI. From the user's perspective, the application would appear to hang. Even worse, if the UI thread were blocked for more than a few seconds (about 5 seconds currently) the user would be presented with the "application not responding" (ANR) dialog. The user might decide to quit your application and uninstall it.

To make sure your app doesn't block the UI thread:

- Complete all work in less than 16 ms for each UI screen.
- Don't run asynchronous tasks and other long-running tasks on the UI thread. Instead, implement tasks on a background thread using `AsyncTask` (for short or interruptible tasks) or `AsyncTaskLoader` (for tasks that are high-priority, or tasks that need to report back to the user or UI).

2. Do UI work only on the UI thread.

Don't use a background thread to manipulate your UI, because the Android UI toolkit is not thread-safe.

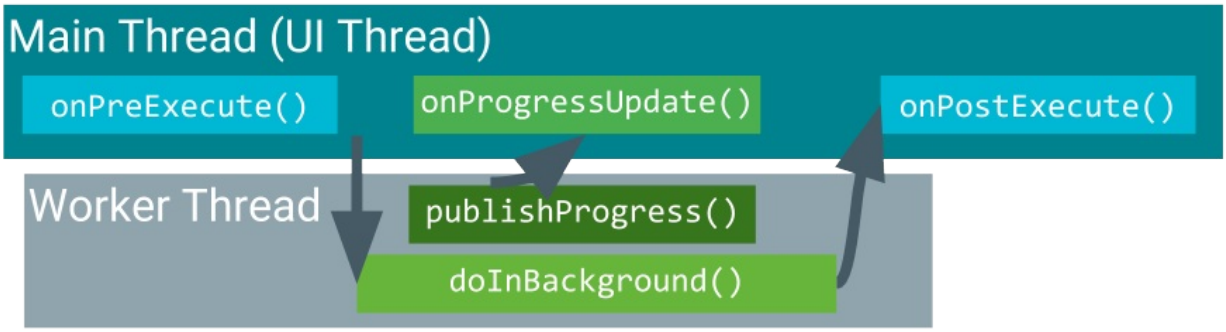
AsyncTask

Use the `AsyncTask` class to implement an asynchronous, long-running task on a worker thread. (A *worker thread* is any thread which is not the main or UI thread.) `AsyncTask` allows you to perform background operations and publish results on the UI thread without manipulating threads or handlers.

When `AsyncTask` is executed, it goes through four steps:

1. `onPreExecute()` is invoked on the UI thread before the task is executed. This step is normally used to set up the task, for instance by showing a progress bar in the UI.
2. `doInBackground(Params...)` is invoked on the background thread immediately after `onPreExecute()` finishes. This step performs a background computation, returns a result, and passes the result to `onPostExecute()`. The `doInBackground()` method can also call `publishProgress(Progress...)` to publish one or more units of progress.
3. `onProgressUpdate(Progress...)` runs on the UI thread after `publishProgress(Progress...)` is invoked. Use `onProgressUpdate()` to report any form of progress to the UI thread while the background computation is executing. For instance, you can use it to pass the data to animate a progress bar or show logs in a text field.
4. `onPostExecute(Result)` runs on the UI thread after the background computation has finished.

For complete details on these methods, see the `AsyncTask reference`. Below is a diagram of their calling order.



AsyncTask usage

To use the `AsyncTask` class, define a subclass of `AsyncTask` that overrides the `doInBackground(Params...)` method (and usually the `onPostExecute(Result)` method as well). This section describes the parameters and usage of `AsyncTask`, then shows a complete example.

AsyncTask parameters

In your subclass of `AsyncTask`, provide the data types for three kinds of parameters:

- "Params" specifies the type of parameters passed to `doInBackground()` as an array.
- "Progress" specifies the type of parameters passed to `publishProgress()` on the background thread. These parameters are then passed to the `onProgressUpdate()` method on the main thread.
- "Result" specifies the type of parameter that `doInBackground()` returns. This parameter is automatically passed to `onPostExecute()` on the main thread.

Specify a data type for each of these parameter types, or use `Void` if the parameter type will not be used. For example:

```
public class MyAsyncTask extends AsyncTask <String, Void, Bitmap>{}
```

In this class declaration:

- The "Params" parameter type is `String`, which means that `MyAsyncTask` takes one or more strings as parameters in `doInBackground()`, for example to use in a query.
- The "Progress" parameter type is `Void`, which means that `MyAsyncTask` won't use the `publishProgress()` or `onProgressUpdate()` methods.
- The "Result" parameter type is `Bitmap`. `MyAsyncTask` returns a Bitmap in `doInbackground()`, which is passed into `onPostExecute()`.

Example of an AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
     protected Long doInBackground(URL... urls) {
         int count = urls.length;
         long totalSize = 0;
         for (int i = 0; i < count; i++) {
             totalSize += Downloader.downloadFile(urls[i]);
             publishProgress((int) ((i / (float) count) * 100));
             // Escape early if cancel() is called
             if (isCancelled()) break;
         }
         return totalSize;
     }

     protected void onProgressUpdate(Integer... progress) {
         setProgressPercent(progress[0]);
     }

     protected void onPostExecute(Long result) {
         showDialog("Downloaded " + result + " bytes");
     }
 }
```

The example above goes through three of the four basic AsyncTask steps:

- `doInBackground()` downloads content, a long-running task. It computes the percentage of files downloaded from the index of the `for` loop and passes it to `publishProgress()`. The check for `isCancelled()` inside the `for` loop ensures that if the task has been cancelled, the system does not wait for the loop to complete.
- `onProgressUpdate()` updates the percent progress. It is is called every time the `publishProgress()` method is called inside `doInBackground()`, which updates the percent progress.
- `doInBackground()` computes the total number of bytes downloaded and returns it. `onPostExecute()` receives the returned result and passes it into `onPostExecute()`, where it is displayed in a dialog.

The parameter types used in this example are:

- `URL` for the "Params" parameter type. The `URL` type means you can pass any number of URLs into the call, and the URLs are automatically passed into the `doInBackground()` method as an array.
- `Integer` for the "Progress" parameter type.
- `Long` for the "Result" parameter type.

Executing an AsyncTask

After you define a subclass of `AsyncTask`, instantiate it on the UI thread. Then call `execute()` on the instance, passing in any number of parameters. (These parameters correspond to the "Params" parameter type discussed above).

For example, to execute the DownloadFilesTask task defined above, use the following line of code:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Cancelling an AsyncTask

You can cancel a task at any time, from any thread, by invoking the `cancel()` method.

- The `cancel()` method returns `false` if the task could not be cancelled, typically because it has already completed normally. Otherwise, `cancel()` returns `true`.
- To find out whether a task has been cancelled, check the return value of `isCancelled()` periodically from `doInBackground(Object[])`, for example from inside a loop as shown in the example above. The `isCancelled()`

method returns `true` if the task was cancelled before it completed normally.

- After an `AsyncTask` task is cancelled, `onPostExecute()` will not be invoked after `doInBackground()` returns. Instead, `onCancelled(Object)` is invoked. The default implementation of `onCancelled(Object)` simply invokes `onCancelled()` and ignores the result.
- By default, in-process tasks are allowed to complete. To allow `cancel()` to interrupt the thread that's executing the task, pass `true` for the value of `mayInterruptIfRunning`.

Limitations of AsyncTask

`AsyncTask` is impractical for some use cases:

- Changes to device configuration cause problems.

  When device configuration changes while an `AsyncTask` is running, for example if the user changes the screen orientation, the activity that created the `AsyncTask` is destroyed and re-created. The `AsyncTask` is unable to access the newly created activity, and the results of the `AsyncTask` aren't published.

- Old `AsyncTask` objects stay around, and your app may run out of memory or crash.

  If the activity that created the `AsyncTask` is destroyed, the `AsyncTask` is not destroyed along with it. For example, if your user exits the application after the `AsyncTask` has started, the `AsyncTask` keeps using resources unless you call `cancel()`.

When to use `AsyncTask`:

- Short or interruptible tasks.
- Tasks that don't need to report back to UI or user.
- Low-priority tasks that can be left unfinished.

For all other situations, use `AsyncTaskLoader`, which is part of the `Loader` framework described next.

Loaders

Background tasks are commonly used to load data such as forecast reports or movie reviews. Loading data can be memory intensive, and you want the data to be available even if the device configuration changes. For these situations, use *loaders*, which are a set of classes that facilitate loading data into an activity.

Loaders use the `LoaderManager` class to manage one or more loaders. `LoaderManager` includes a set of callbacks for when the loader is created, when it's done loading data, and when it's reset.

Starting a loader

Use the `LoaderManager` class to manage one or more `Loader` instances within an activity or fragment. Use `initLoader()` to initialize a loader and make it active. Typically, you do this within the activity's `onCreate()` method. For example:

```
// Prepare the loader.  Either reconnect with an existing one,
// or start a new one.
getLoaderManager().initLoader(0, null, this);
```

If you're using the Support Library, make this call using `getSupportLoaderManager()` instead of `getLoaderManager()`. For example:

```
getSupportLoaderManager().initLoader(0, null, this);
```

The `initLoader()` method takes three parameters:

- A unique ID that identifies the loader. This ID can be whatever you want.

- Optional arguments to supply to the loader at construction, in the form of a `Bundle`. If a loader already exists, this parameter is ignored.
- A `LoaderCallbacks` implementation, which the `LoaderManager` calls to report loader events. In this example, the local class implements the `LoaderManager.LoaderCallbacks` interface, so it passes a reference to itself, `this`.

The `initLoader()` call has two possible outcomes:

- If the loader specified by the ID already exists, the last loader created using that ID is reused.
- If the loader specified by the ID doesn't exist, `initLoader()` triggers the `onCreateLoader()` method. This is where you implement the code to instantiate and return a new loader.
  Note: Whether `initLoader()` creates a new loader or reuses an existing one, the given `LoaderCallbacks` implementation is associated with the loader and is called when the loader's state changes. If the requested loader exists and has already generated data, then the system calls `onLoadFinished()` immediately (during `initLoader()`), so be prepared for this to happen.

  Put the call to `initLoader()` in `onCreate()` so that the activity can reconnect to the *same* loader when the configuration changes. That way, the loader doesn't lose the data it has already loaded.

Restarting a loader

When `initLoader()` reuses an existing loader, it doesn't replace the data that the loader contains, but sometimes you want it to. For example, when you use a user query to perform a search and the user enters a new query, you want to reload the data using the new search term. In this situation, use the `restartLoader()` method and pass in the ID of the loader you want to restart. This forces another data load with new input data.

About the `restartLoader()` method:

- `restartLoader()` uses the same arguments as `initLoader()`.
- `restartLoader()` triggers the `onCreateLoader()` method, just as `initLoader()` does when creating a new loader.
- If a loader with the given ID exists, `restartLoader()` restarts the identified loader and replaces its data.
- If no loader with the given ID exists, `restartLoader()` starts a new loader.

LoaderManager callbacks

The `LoaderManager` object automatically calls `onStartLoading()` when creating the loader. After that, the `LoaderManager` manages the state of the loader based on the state of the activity and data, for example by calling `onLoadFinished()` when the data has loaded.

To interact with the loader, use one of the `LoaderManager callbacks` in the activity where the data is needed:

- Call `onCreateLoader()` to instantiate and return a new loader for the given ID.
- Call `onLoadFinished()` when a previously created loader has finished loading. This is typically the point at which you move the data into activity views.
- Call `onLoaderReset()` when a previously created loader is being reset, which makes its data unavailable. At this point your app should remove any references it has to the loader's data.

The subclass of the `Loader` is responsible for actually loading the data. Which `Loader` subclass you use depends on the type of data you are loading, but one of the most straightforward is `AsyncTaskLoader`, described next. `AsyncTaskLoader` uses an `AsyncTask` to perform tasks on a worker thread.

AsyncTaskLoader

`AsyncTaskLoader` is the loader equivalent of `AsyncTask`. `AsyncTaskLoader` provides a method, `loadInBackground()`, that runs on a separate thread. The results of `loadInBackground()` are automatically delivered to the UI thread, by way of the `onLoadFinished()` `LoaderManager` callback.

AsyncTaskLoader usage

To define a subclass of `AsyncTaskLoader`, create a class that extends `AsyncTaskLoader<D>`, where `D` is the data type of the data you are loading. For example, the following `AsyncTaskLoader` loads a list of strings:

```
public static class StringListLoader extends AsyncTaskLoader<List<String>> {}
```

Next, implement a constructor that matches the superclass implementation:

- Your constructor takes the application context as an argument and passes it into a call to `super()`.
- If your loader needs additional information to perform the load, your constructor can take additional arguments. In the example shown below, the constructor takes a query term.

```
public StringListLoader(Context context, String queryString) {
   super(context);
   mQueryString = queryString;
}
```

To perform the load, use the `loadInBackground()` override method, the corollary to the `doInBackground() method of AsyncTask`. For example:

```
@Override
public List<String> loadInBackground() {
    List<String> data = new ArrayList<String>;
    //TODO: Load the data from the network or from a database
    return data;
}
```

Implementing the callbacks

Use the constructor in the `onCreateLoader()` `LoaderManager` callback, which is where the new loader is created. For example, this `onCreateLoader()` callback uses the `StringListLoader` constructor defined above:

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
   return new StringListLoader(this, args.getString("queryString"));
}
```

The results of `loadInBackground()` are automatically passed into the `onLoadFinished()` callback, which is where you can display the results in the UI. For example:

```
public void onLoadFinished(Loader<List<String>> loader, List<String> data) {
    mAdapter.setData(data);
}
```

The `onLoaderReset()` callback is only called when the loader is being destroyed, so you can leave `onLoaderReset()` blank most of the time, because you won't try to access the data after the loader is destroyed.

When you use `AsyncTaskLoader`, your data survives device-configuration changes. If your activity is permanently destroyed, the loader is destroyed with it, with no lingering tasks that consume system resources.

Loaders have other benefits too, for example they let you monitor data sources for changes and reload the data if a change occurs. You learn more about the specifics of loaders in a future lesson.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Create an AsyncTask

Learn more

- AsyncTask reference
- AsyncTaskLoader reference
- LoaderManager reference
- Processes and Threads
- Loaders guide

7.2: Connect to the Internet

Contents:

- Introduction
- Network security
- Including permissions in the manifest
- Performing network operations on a worker thread
- Making an HTTP connection
- Parsing the results
- Managing the network state
- Related practical
- Learn more

Most Android applications have some data that the user interacts with; it might be news articles, weather information, contacts, game data, user information, and more. Often, this data is provided over the network by a web API.

In this lesson you learn about network security and how to make network calls, which involves these tasks:

1. Include permissions in your AndroidManifest.xml file.
2. On a worker thread, make an HTTP client connection that connects to the network and downloads (or uploads) data.
3. Parse the results, which are usually in JSON format.
4. Check the state of the network and respond accordingly.

Network security

Network transactions are inherently risky, because they involve transmitting data that could be private to the user. People are increasingly aware of these risks, especially when their devices perform network transactions, so it's very important that your app implement best practices for keeping user data secure at all times.

Security best practices for network operations:

- Use appropriate protocols for sensitive data. For example for secure web traffic, use the `HttpsURLConnection` subclass of `HttpURLConnection`.
- Use HTTPS instead of HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on insecure networks such as public Wi-Fi hotspots. Consider using `SSLSocketClass` to implement authenticated, encrypted socket-level communication.
- Don't use localhost network ports to handle sensitive interprocess communication (IPC), because other applications on the device can access these local ports. Instead, use a mechanism that lets you use authentication, for example a `Service`.
- Don't trust data downloaded from HTTP or other insecure protocols. Validate input that's entered into a `WebView` and responses to intents that you issue against HTTP.

For more best practices and security tips, take a look at the Security Tips article.

Including permissions in the manifest

Before your app can make network calls, you need to include a permission in your AndroidManifest.xml file. Add the following tag inside the `<manifest>` tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

When using the network, it's a best practice to monitor the network state of the device so that you don't attempt to make network calls when the network is unavailable. To access the network state of the device, your app needs an additional permission:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Performing network operations on a worker thread

Always perform network operations on a worker thread, separate from the UI. For example, in your Java code you could create an `AsyncTask` (or `AsyncTaskLoader`) implementation that opens a network connection and queries an API. Your main code checks whether a network connection is active. If so, it runs the `AsyncTask` in a separate thread, then displays the results in the UI.

Note: If you run network operations on the main thread instead of on a worker thread, you receive an error.

Making an HTTP connection

Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network. For a refresher on HTTP, visit this Learn HTTP tutorial.

Note: If a web server offers HTTPS, you should use it instead of HTTP for improved security.
The `HttpURLConnection` Android client supports HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. To use the `HttpURLConnection` client, build a URI (the request's destination). Then obtain a connection, send the request and any request headers, download and read the response and any response headers, and disconnect.

Building your URI

To open an HTTP connection, you need to build a request URI. A URI is usually made up of a base URL and a collection of query parameters that specify the resource in question. For example to search for the first five book results for "Pride and Prejudice" in the Google Books API, use the following URI:

```
https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books
```

To construct a request URI programmatically, use the `URI.parse()` method with the `buildUpon()` and `appendQueryParameter()` methods. The following code builds the complete URI shown above:

```
// Base URL for the Books API.
final String BOOK_BASE_URL =  "https://www.googleapis.com/books/v1/volumes?";

final String QUERY_PARAM = "q"; // Parameter for the search string
final String MAX_RESULTS = "maxResults"; // Parameter to limit search results.
final String PRINT_TYPE = "printType"; // Parameter to filter by print type

// Build up the query URI, limiting results to 5 items and printed books.
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
       .appendQueryParameter(QUERY_PARAM, "pride+prejudice")
       .appendQueryParameter(MAX_RESULTS, "5")
       .appendQueryParameter(PRINT_TYPE, "books")
       .build();
```

To convert the URI to a string, use the `toString()` method:

```
String myurl = builtURI.toString();
```

Connect and download data

In the worker thread that performs your network transactions, for example within your override of the `doInBackground()` method in an `AsyncTask`, use the `HttpURLConnection` class to perform an HTTP `GET` request and download the data your app needs. Here's how:

1. To obtain a new `HttpURLConnection`, call `URL.openConnection()` using the URI that you've built. Cast the result to `HttpURLConnection`.

   The URI is the primary property of the request, but request headers can also include metadata such as credentials, preferred content types, and session cookies.

2. Set optional parameters:
   - For a slow connection, you might want a long connection timeout (the time to make the initial connection to the resource) or read timeout (the time to actually read the data).
   - To change the request method to something other than `GET`, use `setRequestMethod()`.
   - If you won't use the network for input, set `setDoInput` to `false`. (Its default is `true`.)
   - For more methods you can set, see the `HttpURLConnection` and `URLConnection` reference documentation.
3. Open an input stream using `getInputStream()`, then read the response and convert it into a string. Response headers typically include metadata such as the response body's content type and length, modification dates, and session cookies. If the response has no body, `getInputStream()` returns an empty stream.
4. Call `disconnect()` to close the connection. Disconnecting releases the resources held by a connection so they can be closed or reused.

These steps are shown in the Request example, below.

If you're posting data over the network and not just receiving data, you need to upload a *request body*, which holds the data to be posted. To do this:

1. Configure the connection so that output is possible by calling `setDoOutput(true)`. (By default, `HttpURLConnection` uses HTTP `GET` requests. When `setDoOutput` is `true`, `HttpURLConnection` uses HTTP `POST` requests by default.)
2. Open an output stream by calling `getOutputStream()`.

For more about posting data to the network, see "Posting Content" in the `HttpURLConnection documentation`.

Note: All network calls must be performed in a worker thread and not on the UI thread.

Request example

The following example sends a request to the URL built in the Building your URI section, above. The request obtains a new `HttpURLConnection`, opens an input stream, reads the response, converts the response into a string, and closes the connection.

```
private String downloadUrl(String myurl) throws IOException {
    InputStream inputStream = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        // Start the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        inputStream = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = convertInputToString(inputStream, len);
        return contentAsString;

    // Close the InputStream and connection
    } finally {
      conn.disconnect();
        if (inputStream != null) {
            inputStream.close();
        }
    }
}
```

Converting the InputStream to a string

An `InputStream` is a readable source of bytes. Once you get an `InputStream`, it's common to decode or convert it into the data type you need. In the example above, the `InputStream` represents plain text from the web page located at https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books.

The `convertInputToString` method defined below converts the `InputStream` to a string so that the activity can display it in the UI. The method uses an `InputStreamReader` instance to read bytes and decode them into characters:

```
// Reads an InputStream and converts it to a String.
public String convertInputToString(InputStream stream, int len)
          throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

Note: If you expect a long response, wrap your `InputStreamReader` inside a `BufferedReader` for more efficient reading of characters, arrays, and lines. For example:
`reader = new BufferedReader(new InputStreamReader(stream, "UTF-8"));`

Parsing the results

When you make web API queries, the results are often in JSON format.

Below is an example of a JSON response from an HTTP request. It shows the names of three menu items in a popup menu and the methods that are triggered when the menu items are clicked:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
```

To find the value of an item in the response, use methods from the `JSONObject` and `JSONArray` classes. For example, here's how to find the `"onclick"` value of the third item in the `"menuitem"` array:

```
JSONObject data = new JSONObject(responseString);
JSONArray menuItemArray = data.getJSONArray("menuitem");
JSONObject thirdItem = menuItemArray.getJSONObject(2);
String onClick = thirdItem.getString("onclick");
```

Managing the network state

Making network calls can be expensive and slow, especially if the device has little connectivity. Being aware of the network connection state can prevent your app from attempting to make network calls when the network isn't available.

Sometimes it's also important for your app to know what kind of connectivity the device has: Wi-Fi networks are typically faster than data networks, and data networks are often metered and expensive. To control when certain tasks are performed, monitor the network state and respond appropriately. For example, you may want to wait until the device is connected to Wi-Fi to perform a large file download.

To check the network connection, use the following classes:

- `ConnectivityManager` answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.
- `NetworkInfo` describes the status of a network interface of a given type (currently either mobile or Wi-Fi).

The following code snippet tests whether Wi-Fi and mobile are connected. In the code:

- The `getSystemService` method gets an instance of `ConnectivityManager`.
- The `getNetworkInfo` method gets the status of the device's Wi-Fi connection, then its mobile connection. The `getNetworkInfo` method returns a `NetworkInfo` object, which contains information about the given network's connection status (whether it's idle, connecting, and so on).
- The `networkInfo.isConnected()` method returns `true` if the given network is connected. If the network is connected, it can be used to establish sockets and pass data.

```
private static final String DEBUG_TAG = "NetworkStatusExample";
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Connect to the Internet with AsyncTask and AsyncTaskLoader

Learn more

- Connecting to the Network
- Managing Network Usage
- HttpURLConnection reference
- ConnectivityManager reference
- InputStream reference

7.3: Broadcast Receivers

Contents:

- Introduction
- Broadcast intents
- Broadcast receivers
- Security guidelines
- LocalBroadcastManager
- Related practical
- Learn more

Explicit intents are used to start specific, fully qualified activities, as well as to pass information between activities in your app. *Implicit intents* are used to start activities based on registered components that the system is aware of, for example general functionality.

In this lesson you learn about *broadcast intents*, which don't start activities but instead are delivered to *broadcast receivers*.

Broadcast intents

The intents you've seen up to now always resulted in an activity being launched, either a specific activity from your application or an activity from a different application that could fulfill the requested action. But sometimes an intent doesn't have a specific recipient, and sometimes you don't want an activity to be launched in response to an intent. For example, when your app receives a system intent indicating that the network state of a device has changed, you probably don't want to launch an activity, but you may want to disable some functionality of your app.

For this reason there's a third type of intent that can be delivered to any interested application: the *broadcast intent*. Although broadcast intents are defined the same way as implicit intents, each type of intent has important distinguishing characteristics:

- Broadcast intents are delivered using `sendBroadcast()` or a related method, while other types of intents use `startActivity()` to start activities. When you broadcast an intent, you never find or start an activity. Likewise, there's no way for a broadcast receiver to see or capture intents used with `startActivity()`.
- A broadcast intent is a background operation that the user is not normally aware of. Starting an activity with an intent, on the other hand, is a foreground operation that modifies what the user is currently interacting with.

There are two types of broadcast intents, those delivered by the system (system broadcast intents), and those that your app delivers (custom broadcast intents).

System broadcast intents

The system delivers a *system broadcast intent* when a system event occurs that might interest your app. For example:

- When the device boots, the system sends an `ACTION_BOOT_COMPLETED` system broadcast intent. This intent contains the constant value `"android.intent.action.BOOT_COMPLETED"`.
- When the device is connected to external power, the system sends `ACTION_POWER_CONNECTED`, which contains the constant value `"android.intent.action.ACTION_POWER_CONNECTED"`. When the device is disconnected from external power, the system sends `ACTION_POWER_DISCONNECTED`.
- When the device is low on memory, the system sends `ACTION_DEVICE_STORAGE_LOW`. This intent contains the constant value `"android.intent.action.DEVICE_STORAGE_LOW"`.

`ACTION_DEVICE_STORAGE_LOW` is a *sticky* broadcast, which means that the broadcast value is held in a cache. If you need to know whether your broadcast receiver is processing a value that's in the cache (sticky) or a value that's being broadcast in the present moment, use `isInitialStickyBroadcast()`.

For more about common system broadcasts, visit the `Intent` reference.

To receive system broadcast intents, you need to create a broadcast receiver.

Custom broadcast intents

Custom broadcast intents are broadcast intents that your application sends out. Use a custom broadcast intent when you want your app to take an action without launching an activity, for example when you want to let other apps know that data has been downloaded to the device and is available for them to use.

To create a custom broadcast intent, create a custom intent action. To deliver a custom broadcast to other apps, pass the intent to `sendBroadcast()`, `sendOrderedBroadcast()`, or `sendStickyBroadcast()`. (For details about these methods, see the `Context` reference documentation.)

For example, the following method creates an intent and broadcasts it to all interested broadcast receivers:

```
public void sendBroadcastIntent() {
   Intent intent = new Intent();
   intent.setAction("com.example.myproject.ACTION_SHOW_TOAST");
   sendBroadcast(intent);
}
```

Note: When you specify the action for the intent, use your unique package name (`com.example.myproject` in the example) to make sure your intent doesn't conflict with an intent that is broadcast from a different app or from the system.

Broadcast receivers

Broadcast intents aren't targeted at specific recipients. Instead, interested apps register a component to "listen" for these kind of intents. This listening component is called a *broadcast receiver*.

Use broadcast receivers to respond to messages that are broadcast from other apps or from the system. To create a broadcast receiver:

1. Define a subclass of the `BroadcastReceiver` class and implement the `onReceive()` method.
2. Register the broadcast receiver either dynamically in Java, or statically in your app's manifest file.

   As part of this step, use an *intent filter* to specify which kinds of broadcast intents you're interested in receiving.

These steps are described below.

Define a subclass of BroadcastReceiver

To create a broadcast receiver, define a subclass of the `BroadcastReceiver` class. This subclass is where intents are delivered (if they match the intent filters you set when you register the subclass, which happens in a later step).

Within your subclass definition:

- Implement the required `onReceive()` method.
- Include any other logic that your broadcast receiver needs.

Example: Create a broadcast receiver

In this example, the `AlarmReceiver` subclass of `BroadcastReceiver` shows a Toast message if the incoming broadcast intent has the action `ACTION_SHOW_TOAST`:

```
private class AlarmReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
       if (intent.getAction().equals(ACTION_SHOW_TOAST)) {
            CharSequence text = "Broadcast Received!";
            int duration = Toast.LENGTH_SHORT;

            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
       }
   }
}
```

Note: Don't use asynchronous operations in your `onReceive()` implementation, because once your code returns from `onReceive()`, the system considers the `BroadcastReceiver` object to be finished. If `onReceive()` were to start an asynchronous operation, the system would kill the `BroadcastReceiver` process before the asynchronous operation had a chance to complete.
If you need a long-running operation that doesn't require a UI, use a Service launched from the broadcast receiver. In particular:

- You can't show a dialog from within a `BroadcastReceiver`. Instead, use the `NotificationManager` API.
- You can't bind to a service from within a `BroadcastReceiver`. Instead, use `Context.startService()` to send a command to the service.

Registering your broadcast receiver and setting intent filters

There are two ways to register your broadcast receiver: statically in the manifest, or dynamically in your activity.

Static registration

To register your broadcast receiver statically, add a `<receiver>` element to your AndroidManifest.xml file. Within the `<receiver>` element:

- Use the path to your `BroadcastReceiver` subclass as the `android:name` attribute.
- To prevent other applications from sending broadcasts to your receiver, set the optional `android:exported` attribute to `false`. This is an important security guideline.
- To specify the types of intents the component is listening for, use a nested `<intent-filter>` element.

Example: Static registration and intent filter for a custom broadcast intent

The following code snippet is an example of static registration for a broadcast receiver that listens for a custom broadcast intent with " `ACTION_SHOW_TOAST` " in the name of its action:

- The receiver's `name` is the module name plus the name of the `BroadcastReceiver` subclass defined above (`AlarmReceiver`).
- The receiver is not exported, meaning that no other applications can deliver broadcasts to this app.
- The receiver includes an intent filter that checks whether incoming intents include an action named `ACTION_SHOW_TOAST`.

```
<receiver
android:name="com.example.myproject.AlarmReceiver"
android:exported="false">
<intent-filter>
   <action android:name="com.example.myproject.intent.action.ACTION_SHOW_TOAST"/>
</intent-filter>
</receiver>
```

Intent filters

When the system receives an implicit intent to start an activity, it searches for the best activity for the intent by comparing it to intent filters, based on three aspects:

- Action: Does the action specified in the intent match one of the `<action>` names listed in the filter? In the example above, only intents with `ACTION_SHOW_TOAST` in the name of their action match the filter.
- Data: Does the data in the intent match one of the `<data>` types that are listed in the filter?
- Category: Does every category in the intent match a `<category>` that's named in the filter?

Example: Intent filter for a system broadcast intent

This example shows an intent filter for a receiver that listens for the device to finish booting:

```
<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
</intent-filter>
```

To learn more about using intent filters to select intents, see the Intent Resolution section of the intent guide.

If no intent filters are specified, the broadcast receiver can only be activated with an explicit broadcast intent that names the component by name. (This is similar to how you launch activities by their class name with explicit intents.)

If you use static registration for your broadcast receiver, the Android system creates a new process to run your broadcast receiver if no processes associated with your application are running. This means that the receiver will respond, even if your app is not running.

Dynamic registration and unregistration

You can also register a broadcast receiver dynamically, which ties its operation to the lifecycle of your activity. To register your receiver dynamically, call `registerReceiver()` and pass in the `BroadcastReceiver` object and an intent filter. For example:

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(ACTION_SHOW_TOAST);

mReceiver = new AlarmReceiver();
registerReceiver(mReceiver, intentFilter);
```

Note: If you register your receiver to receive only local broadcasts, you must register it dynamically; static registration isn't an option.
You also need to unregister the receiver by calling `unregisterReceiver()` and passing in your `BroadcastReceiver` object. For example:

```
unregisterReceiver(mReceiver);
```

Where you call these methods depends on the desired lifecycle of your `BroadcastReceiver` object:

- If the receiver is only needed when your activity is visible (for example, to disable a network function when the network is not available), then register the receiver in `onResume()`. Unregister the receiver in `onPause()`.
- You can also use the `onStart()` / `onStop()` or `onCreate()/onDestoy()` method pairs, if they are more appropriate for your use case.

Security guidelines

When you use broadcast intents and broadcast receivers, information is sent between applications, which creates security risks. To avoid these risks, you can use `LocalBroadcastManager` (described below), or you can follow these guidelines:

- Make sure that the names of intent actions and other strings are in a namespace that you own, or else you may inadvertently conflict with other applications. The intent namespace is global.
- When you use `registerReceiver()`, any application can send broadcasts to that registered receiver. To control who can send broadcasts to it, use the permissions described below.
- When you register a broadcast receiver statically, any other application can send broadcasts to it, regardless of the filters you specify. To prevent others from sending to it, make it unavailable to them with `android:exported="false"`.
- When you use `sendBroadcast()` or related methods, any other application can receive your broadcasts. To control who can receive such broadcasts, use the permissions described below.

Either the sender or receiver of a broadcast can enforce access permissions:

- To enforce a permission when **sending** a broadcast, supply a non-null permission argument to `sendBroadcast()`.

  Only receivers who have been granted this permission (by requesting it with the `<uses-permission>` tag in their AndroidManifest.xml) can receive the broadcast.

- To enforce a permission when **receiving** a broadcast, supply a non-null permission when registering your receiver either when calling `registerReceiver()` or in the static `<receiver>` tag in your AndroidManifest.xml.

  Only broadcasters who have been granted this permission (by requesting it with the `<uses-permission>` tag in their AndroidManifest.xml) will be able to send an intent to the receiver. The receiver has to request permission in the manifest, regardless of whether the sender is registered statically or dynamically.

LocalBroadcastManager

To avoid having to manage the security aspects described in the Security guidelines, use the `LocalBroadcastManager` class. `LocalBroadcastManager` lets you send and receive broadcasts within a single process and a single application, which means you don't have to worry about cross-application security.

Sending local broadcasts

To send a broadcast using `LocalBroadcastManager`:

1. Get an instance of `LocalBroadcastManager` by calling `getInstance()` and passing in the application context.
2. Call `sendBroadcast()` on the instance, passing in the intent that you want to broadcast.

For example:

```
LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent);
```

Registering your receiver for local broadcasts

To register your receiver to receive only local broadcasts:

1. Get an instance of `LocalBroadcastManager` by calling `getInstance()` and passing in the application context.
2. Call `registerReceiver()`, passing in the receiver and an intent filter as you would for a regular broadcast receiver. You must register local receivers dynamically, because static registration in the manifest is unavailable.

For example:

```
LocalBroadcastManager.getInstance(this).registerReceiver
       (mReceiver, new IntentFilter(CustomReceiver.ACTION_CUSTOM_BROADCAST));
```

To unregister the broadcast receiver:

```
LocalBroadcastManager.getInstance(this).unregisterReceiver(mReceiver);
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Broadcast Receivers

Learn more

- BroadcastReceiver reference
- Intents and Intent Filters guide
- LocalBroadcastManager reference

7.4: Services

Contents:

- Introduction
- What is a service?
- Declaring services in the manifest
- Started services
- Bound services
- Service lifecycle
- Foreground services
- Scheduled services
- Learn More

In this chapter you learn about the different types of services, how to use them, and how to manage their lifecycles within your app.

What is a service?

A *service* is an application component that performs long-running operations, usually in the background. A service doesn't provide a user interface (UI). (An *activity*, on the other hand, provides a UI.)

A service can be *started*, *bound*, or both:

- A *started service* is a service that an application component starts by calling `startService()`.

  Use started services for tasks that run in the background to perform long-running operations. Also use started services for tasks that perform work for remote processes.

- A *bound service* is a service that an application component binds to itself by calling `bindService()`.

  Use bound services for tasks that another app component interacts with to perform interprocess communication (IPC). For example, a bound service might handle network transactions, perform file I/O, play music, or interact with a content provider.

Note: A service runs in the main thread of its hosting process—the service doesn't create its own thread and doesn't run in a separate process unless you specify that it should.

If your service is going to do any CPU-intensive work or blocking operations (such as MP3 playback or networking), create a new thread within the service to do that work. By using a separate thread, you reduce the risk of Application Not Responding (ANR) errors, and the application's main thread can remain dedicated to user interaction with your activities. To implement any kind of service in your app:

1. Declare the service in the manifest.
2. Create implementation code, as described in Started services and Bound services, below.
3. Manage the service lifecycle.

Declaring services in the manifest

As with activities and other components, you must declare all services in your application's manifest file. To declare a service, add a `<service>` element as a child of the `<application>` element. For example:

```
<manifest ... >
  ...
  <application ... >
      <service android:name="ExampleService"
               android:exported="false" />
      ...
  </application>
</manifest>
```

To block access to a service from other applications, declare the service as private. To do this, set the `android:exported` attribute to `false`. This stops other apps from starting your service, even when they use an explicit intent.

Started services

How a service starts:

1. An application component such as an activity calls `startService()` and passes in an `Intent`. The `Intent` specifies the service and includes any data for the service to use.
2. The system calls the service's `onCreate()` method and any other appropriate callbacks on the main thread. It's up to the service to implement these callbacks with the appropriate behavior, such as creating a secondary thread in which to work.
3. The system calls the service's `onStartCommand()` method, passing in the `Intent` supplied by the client in step 1. (The *client* in this context is the application component that calls the service.)

Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself by calling `stopSelf()`, or another component can stop it by calling `stopService()`.

For instance, suppose an activity needs to save data to an online database. The activity starts a companion service by passing an `Intent` to `startService()`. The service receives the intent in `onStartCommand()`, connects to the Internet, and performs the database transaction. When the transaction is done, the service uses `stopSelf()` to stop itself and is destroyed. (This is an example of a service you want to run in a worker thread instead of the main thread.)

IntentService

Most started services don't need to handle multiple requests simultaneously, and if they did it could be a dangerous multi-threading scenario. For this reason, it's probably best if you implement your service using the `IntentService` class.

`IntentService` is a useful subclass of `Service`:

- `IntentService` automatically provides a worker thread to handle your `Intent`.
- `IntentService` handles some of the boilerplate code that regular services need (such as starting and stopping the service).
- `IntentService` can create a work queue that passes one intent at a time to your `onHandleIntent()` implementation, so you don't have to worry about multi-threading.

To implement `IntentService`:

1. Provide a small constructor for the service.
2. Create an implementation of `onHandleIntent()` to do the work that the client provides.

Here's an example implementation of `IntentService`:

```
public class HelloIntentService extends IntentService {
  /**
   * A constructor is required, and must call the super IntentService(String)
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

Bound services

A service is "bound" when an application component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, and get results, sometimes using interprocess communication (IPC) to send and receive information across processes. A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

A bound service generally does not allow components to start it by calling `startService()`.

Implementing a bound service

To implement a bound service, define the interface that specifies how a client can communicate with the service. This interface, which your service returns from the `onBind()` callback method, must be an implementation of `IBinder`.

To retrieve the `IBinder` interface, a client application component calls `bindService()`. Once the client receives the `IBinder`, the client interacts with the service through that interface.

There are multiple ways to implement a bound service, and the implementation is more complicated than a started service. For complete details about bound services, see Bound Services.

Binding to a service

To bind to a service that is declared in the manifest and implemented by an app component, use `bindService()` with an explicit `Intent`.

Caution: Do not use an implicit intent to bind to a service. Doing so is a security hazard, because you can't be certain what service will respond to your intent, and the user can't see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `bindService()` with an implicit `Intent`.

Service lifecycle

The lifecycle of a service is simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed. Because a service has no UI, services can continue to run in the background with no way for the user to know, even if the user switches to another application. This consumes resources and drains battery.

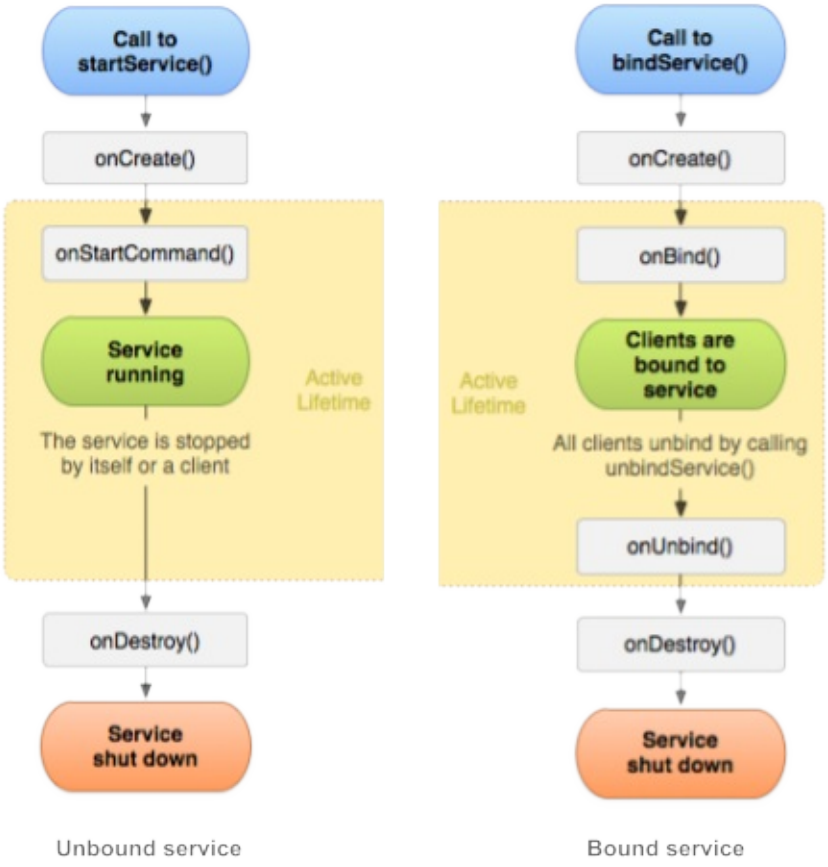
Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

```
public class ExampleService extends Service {
    int mStartMode;       // indicates how to behave if the service is killed
    IBinder mBinder;      // interface for clients that bind
    boolean mAllowRebind; // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
        return mBinder;
    }
    @Override
    public boolean onUnbind(Intent intent) {
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }
    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }
    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

Lifecycle of started services vs. bound services

A bound service exists only to serve the application component that's bound to it, so when no more components are bound to the service, the system destroys it. Bound services don't need to be explicitly stopped the way started services do (using `stopService()` or `stopSelf()`).

The diagram below shows a comparison between the started and bound service lifecycles.



Foreground services

While most services run in the background, some run in the foreground. A *foreground service* is a service that the user is aware of, so it's not a candidate for the system to kill when low on memory.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.

To request that a service run in the foreground, call `startForeground()` instead of `startService()`. This method takes two parameters: an integer that uniquely identifies the notification and the `Notification` for the status bar. This notification is *ongoing*, meaning that it can't be dismissed. It stays in the status bar until the service is stopped or removed from the foreground.

For example:

```
NotificationCompat.Builder mBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!");
startForeground(ONGOING_NOTIFICATION_ID, mBuilder.build());
```

Note: The integer ID you give to `startForeground()` must not be 0.
To remove the service from the foreground, call `stopForeground()`. This method takes a boolean, indicating whether to remove the status bar notification. This method doesn't stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

Scheduled services

For API level 21 and higher, you can launch services using the `JobScheduler` API. To use `JobScheduler`, you need to register jobs and specify their requirements for network and timing. The system schedules jobs for execution at appropriate times.

The `JobScheduler` interface provides many methods to define service-execution conditions. For details, see the `JobScheduler reference`.

Learn more

- Services guide
- Running in a Background Service

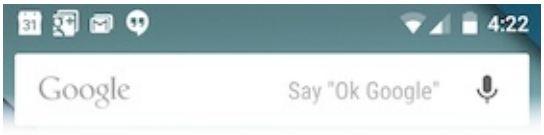
8.1: Notifications

Contents:

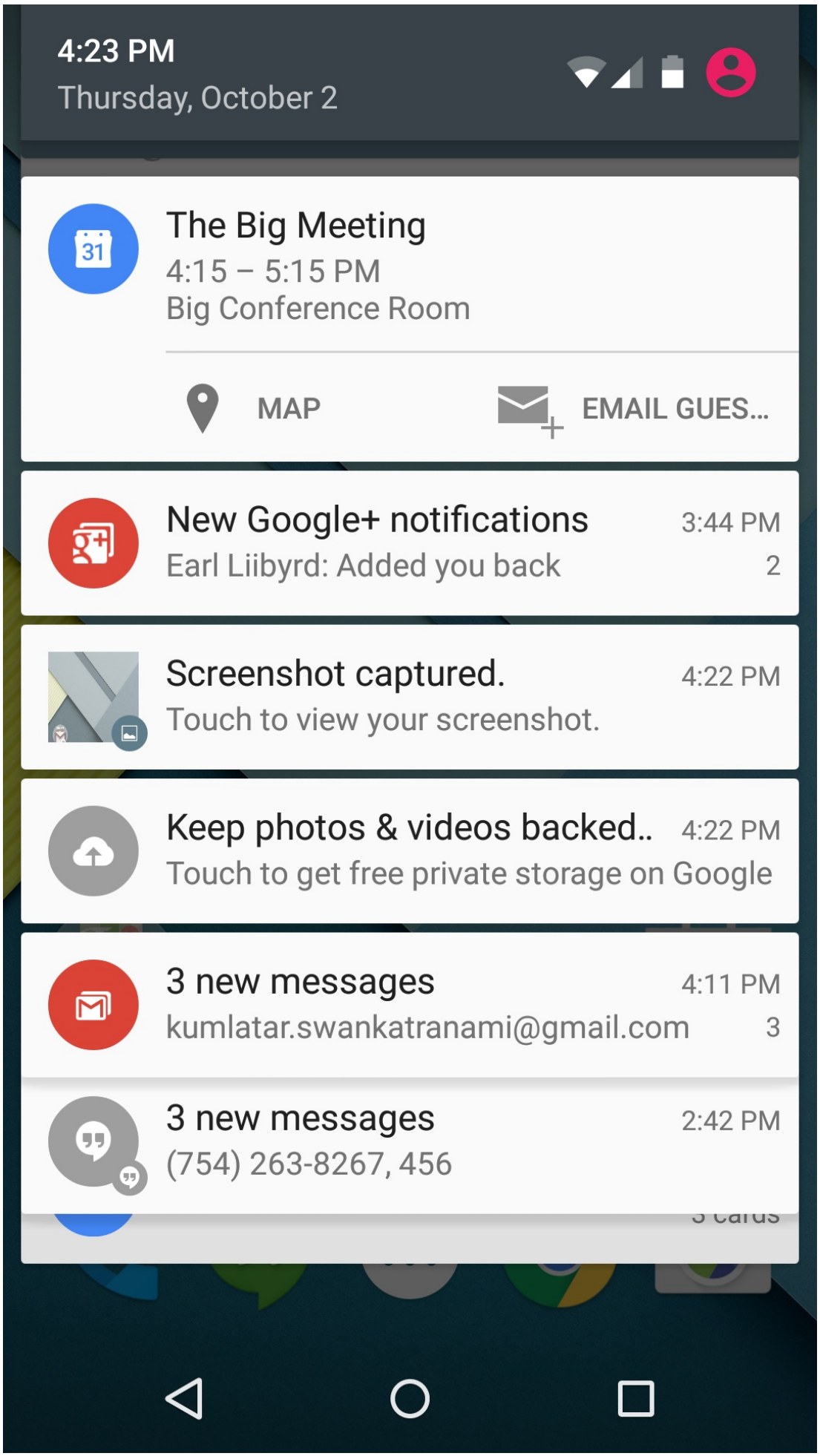
- Introduction
- What is a notification?
- Creating notifications
- Delivering notifications
- Reusing notifications
- Clearing notifications
- Notification compatibility
- Notification design guidelines
- Related practical
- Learn more

In this chapter you learn how to create, deliver, and reuse notifications, and how to make them compatible for different Android versions.

What is a notification?

A *notification* is a message your app displays to the user outside your application's normal UI. When you tell the system to issue a notification, the notification first appears to the user as an icon in the *notification area*, on the left side of the status bar. 

To see the details of the notification, the user opens the *notification drawer*, or views the notification on the lock screen if the device is locked. The notification area, the lock screen, and the notification drawer are system-controlled areas that the user can view at any time.


4:23 PM
Thursday, October 2
The Big Meeting
4:15 – 5:15 PM
Big Conference Room
MAP
EMAIL GUES...
New Google+ notifications
3:44 PM
Earl Liibyrd: Added you back
2
Screenshot captured.
4:22 PM
Touch to view your screenshot.
Keep photos & videos backed..
4:22 PM
Touch to get free private storage on Google
3 new messages
4:11 PM
kumlatar.swankatranami@gmail.com
3
3 new messages
2:42 PM
(754) 263-8267, 456

The screenshot shows an "open" notification drawer. The the status bar isn't visible, because the notification drawer is open.

This process is described below.

Creating notifications

You create a notification using the `NotificationCompat.Builder` class. (Use `NotificationCompat` for the best backward compatibility. For more information, see Notification compatibility.) The builder classes simplify the creation of complex objects.

To create a `NotificationCompat.Builder`, pass the application context to the constructor:

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);
```

Setting notification components

When using NotificationCompat.Builder, you must assign a small icon, text for a title, and the notification message. You should keep the notification message shorter than 40 characters and not repeat what's in the title. For example:

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("Dinner is ready!")
    .setContentText("Lentil soup, rice pilaf, and cake for dessert.");
```

You also need to set an `Intent` that determines what happens when the user clicks the notification. Usually this Intent results in your app launching an Activity.

To make sure the system delivers the `Intent` even when your app isn't running when the user clicks the notification, wrap the `Intent` in a `PendingIntent` object, which allows the system to deliver the `Intent` regardless of the app state.

To instantiate a `PendingIntent`, use one of the following methods, depending on how you want the contained `Intent` to be delivered:

- To launch an Activity when a user clicks on the notification, use `PendingIntent.getActivity()`, passing in an explicit `Intent` for the Activity you want to launch. The `getActivity()` method corresponds to an `Intent` delivered using `startActivity()`.
- For an `Intent` passed into `startService()` (for example a service to download a file), use `PendingIntent.getService()`.
- For a broadcast `Intent` delivered with `sendBroadcast()`, use `PendingIntent.getBroadcast()`.

Each of these `PendingIntent` methods take the following arguments:

- The application context.
- A request code, which is a constant integer ID for the `PendingIntent`.
- The `Intent` to be delivered.
- A `PendingIntent` flag that determines how the system handles multiple `PendingIntent` objects from the same application.

For example:

```
Intent contentIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingContentIntent = PendingIntent.getActivity(this, 0,
    contentIntent, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(pendingContentIntent);
```

To learn more about `PendingIntent`, see the `PendingIntent documentation`.

Optional components

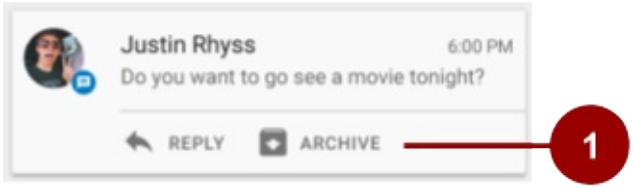
You can use various options with notifications, including:

- Notification actions
- Priorities
- Expanded layouts
- Ongoing notifications

For other options you can use with notifications, see the `NotificationCompat.Builder` reference.

Notification actions

A *notification action* is an action that the user can take on the notification. The action is made available via an action button on the notification. Like the `Intent` that determines what happens when the user clicks the notification, a notification action uses a `PendingIntent` to complete the action. The Android system usually displays a notification action as a button adjacent to the notification content. Starting with Android 4.1 (API level 16), notifications support icons embedded below the content text, as shown in the screenshot below.



1. This notification has two actions that the user can take, "Reply," or "Archive." Each has an icon.

To add a notification action, use the `addAction()` method with the `NotificationCompat.Builder` object. Pass in the icon, the title string and the `PendingIntent` to trigger when the user taps the action. For example:

```
mBuilder.addAction(R.drawable.car, "Get Directions", mapPendingIntent);
```

To ensure that an action button's functionality is always available, follow the instructions in the Notification compatibility section, below.

Notification priority

Android allows you to assign a priority level to each notification to influence how the Android system will deliver it. Notifications have a priority between `MIN` (`-2`) and `MAX` (`2`) that corresponds to their importance. The following table shows the available priority constants defined in the `Notification` class.

Priority Constant	Use
PRIORITY_MAX	For critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a time-critical task.
PRIORITY_HIGH	Primarily for important communication, such as messages or chats.
PRIORITY_DEFAULT	For all notifications that don't fall into any of the other priorities described here.
PRIORITY_LOW	For information and events that are valuable or contextually relevant, but aren't urgent or time-critical.
PRIORITY_MIN	For nice-to-know background information. For example, weather or nearby places of interest.

To change the priority of a notification, use the `setPriority()` method on the `NotificationCompat.Builder` object, passing in one of the above constants.

```
mBuilder.setPriority(Notification.PRIORITY_HIGH);
```

Notifications can be intrusive. Using notification priority correctly is the first step in making sure that your users don't uninstall your app because it's too distracting.

Peeking

Notifications with a priority of `HIGH` or `MAX` can *peek*, which means they slide briefly into view on the user's current screen, no matter what apps the user is using. Note that on devices running Android 6.0 and higher, users can block peeking by changing the device's "App notification" settings. This means you can't rely on notifications peeking, even if you set them up that way.

To create a notification that can peek:

1. Set the priority to `HIGH` or `MAX`.
2. Set a sound or light pattern using the `setDefaults()` method on the builder, passing the `DEFAULTS_ALL` constant. This gives the notification a default sound, light pattern, and vibration.

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("My notification")
    .setContentText("Hello World!")
    .setPriority(PRIORITY_HIGH)
    .setDefaults(DEFAULTS_ALL);
```

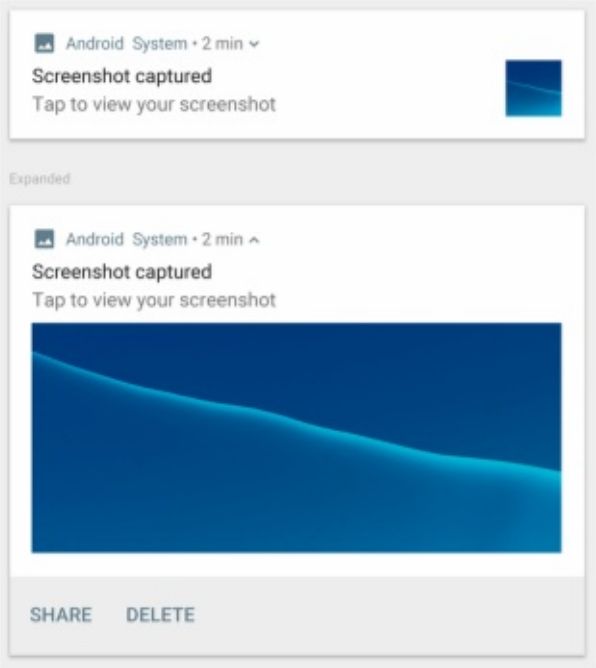
Expanded view layouts

Notifications in the notification drawer appear in two main layouts, *normal view* (which is the default) and *expanded view*. Expanded view notifications were introduced in Android 4.1. Use them sparingly, because they take up more space and attention than normal view layouts.

To create notifications that appear in an expanded layout, use one of these helper classes:

- Use `NotificationCompat.BigTextStyle` for large-format notifications that include a lot of text.
- Use `NotificationCompat.InboxStyle` for large-format notifications that include a list of up to five strings.
- Use `Notification.MediaStyle` for media playback notifications. There is currently no `NotificationCompat` version of this style, so it can only be used on devices with Android 4.1 or above. See the Notification compatibility section for more information.

- Use `NotificationCompat.BigPictureStyle`, shown in the screenshot below, for large-format notifications that include a large image attachment.



For example, here's how you'd set the `BigPictureStyle` on a notification:

```
NotificationCompat notif = new NotificationCompat.Builder(mContext)
    .setContentTitle("New photo from " + sender.toString())
    .setContentText(subject)
    .setSmallIcon(R.drawable.new_post)
    .setLargeIcon(aBitmap)
    .setStyle(new NotificationCompat.BigPictureStyle()
        .bigPicture(aBigBitmap)
        .setBigContentTitle("Large Notification Title"))
    .build();
```

To learn more about implementing expanded styles, see the NotificationCompat.Style documentation.

Ongoing notifications

Ongoing notifications are notifications that can't be dismissed by the user. Your app must explicitly cancel them by calling `cancel()` or `cancelAll()`. Creating multiple ongoing notifications is a nuisance to your users since they are unable to cancel the notification. Use ongoing notifications sparingly.

To make a notification ongoing, set `setOngoing()` to `true`. Use ongoing notifications to indicate background tasks that the user actively engages with (such as playing music) or tasks that occupy the device (such as file downloads, sync operations, and active network connections).

Delivering notifications

Use the `NotificationManager` class to deliver notifications:

1. Call `getSystemService()`, passing in the `NOTIFICATION_SERVICE` constant, to create an instance of `NotificationManager`.
2. Call `notify()` to deliver the notification. In the `notify()` method, pass in these two values:
   - A notification ID, which is used to update or cancel the notification.
   - The `NotificationCompat` object that you created using the `NotificationCompat.Builder` object.

The following example creates a `NotificationManager` instance, then builds and delivers a notification:

```
mNotifyManager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);

//Builds the notification with all the parameters
NotificationCompat.Builder notifyBuilder = new NotificationCompat.Builder(this)
       .setContentTitle(getString(R.string.notification_title))
       .setContentText(getString(R.string.notification_text))
       .setSmallIcon(R.drawable.ic_android)
       .setContentIntent(notificationPendingIntent)
       .setPriority(NotificationCompat.PRIORITY_HIGH)
       .setDefaults(NotificationCompat.DEFAULT_ALL);

//Delivers the notification
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

Reusing notifications

When you need to issue a notification multiple times for the same type of event, you can update a previous notification by changing some of its values, adding to it, or both.

To reuse an existing notification:

1. Update a `NotificationCompat.Builder` object and build a `Notification` object from it, as when you first created and built the notification.
2. Deliver the notification with the same ID you used previously.
   Important: If the previous notification is still visible, the system updates it from the contents of the `Notification` object. If the previous notification has been dismissed, a new notification is created.

Clearing notifications

Notifications remain visible until one of the following happens:

- If the notification can be cleared, it disappears when the user dismisses it individually or by using "Clear All."
- If you called `setAutoCancel()` when you created the notification, the notification disappears when the user clicks it.
- If you call `cancel()` for a specific notification ID, the notification disappears.
- If you call `cancelAll()`, all the notifications you've issued disappear.

Because ongoing notifications can't be dismissed by the user, your app must cancel them by calling `cancel()` or `cancelAll()`.

Notification compatibility

To ensure the best compatibility, create notifications with `NotificationCompat` and its subclasses, particularly `NotificationCompat.Builder`.

Keep in mind that not all notification features are available for every Android version, even though the methods to set them are in the support library class `NotificationCompat.Builder`. For example, expanded view layouts for notifications are only available on Android 4.1 and higher, but action buttons depend on expanded view layouts. This means that if you use notification action buttons, they don't show up on devices running anything before Android 4.1.

To solve this:

- Don't rely on notification action buttons to carry out a notification's action; instead make the action available in an Activity. You may want to add a new Activity to do this.

For example, if you set a notification action that provides a control to stop and start media playback, first implement this control in an Activity in your app.

- Have the Activity start when users click the notification. To do this:
  1. Create a `PendingIntent` for the Activity.
  2. Call `setContentIntent()` to add the `PendingIntent` to the notification.
- Use `addAction()` to add features to the notification as needed. Remember that any functionality you add also has to be available in the Activity that starts when users click the notification.

Notification design guidelines

Notifications always interrupt the user. As such they must be short, timely, and most of all, relevant.

- **Relevant:** Ask yourself whether this information is essential for the user. What happens if they don't get the notification? For example, scheduled calendar events are likely relevant.
- **Timely:** Notifications need to appear when they are useful. For example, notifying the user when it's time to leave for an appointment is useful.
- **Short:** Use as few words as possible. Now, challenge yourself to say it with fewer.

Give users the power to choose:

- Provide settings in your app that allow users to choose the kinds of notifications they want to receive, and how they want to receive them.

In addition to these basic principles, notifications have their own design guidelines:

- To learn how to design notifications and their interactions, see the Material Design notification patterns documentation.
- To learn how to design notifications and their interactions for older Android versions, see Notifications, Android 4.4 and Lower.
- For important details about Material Design changes introduced in Android 5.0 (API level 21), see the Material Design training.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Notifications

Learn more

Guides

- Notifications
- Notification design guide

Reference

- NotificationCompat.Builder reference
- NotificationCompat.Style reference

8.2: Scheduling Alarms

Contents:

- Introduction
- Alarm types
- Alarm best practices
- Scheduling an alarm
- Checking for an existing alarm
- Canceling an alarm
- User-visible alarms ("alarm clocks")
- Related practical
- Learn more

You already know how to use broadcast receivers to make your app respond to system events even when your app isn't running. In this chapter, you'll learn how to use alarms to schedule tasks for specific times, whether or not your app is running at the time the alarm is set to go off. Alarms can either be single use or repeating. For example, you can use a repeating alarm to schedule a download every day at the same time.

To create alarms, you use the `AlarmManager` class. Alarms in Android have the following characteristics:

- Alarms let you send intents at set times or intervals. You can use alarms with broadcast receivers to start services and perform other operations.
- Alarms operate outside your app, so you can use them to trigger events or actions even when your app isn't running, and even if the device is asleep.
- When used correctly, alarms can help you minimize your app's resource requirements. For example, you can schedule operations without relying on timers or continuously running background services.

When *not* to use an alarm:

- For timing events such as ticks and timeouts, and for timed operations that are guaranteed to happen during the lifetime of your app, use the `Handler` class with `Timer` and `Thread`. This approach gives Android better control over system resources than if you used alarms.
- For server sync operations, use `SyncAdapter` with the Google Cloud Messaging Service.
- For tasks that can wait until conditions are favorable, such as when the device is connected to WiFi and is charging (for example, updating weather information or news stories), you might not want to use alarms. For these tasks on API 21+ devices, consider using `JobScheduler`, which you will learn about in an upcoming lesson.

Alarm types

There are two general types of alarms in Android: *elapsed real-time alarms* and *real-time clock (RTC)* alarms, and both use `PendingIntent` objects.

Elapsed real-time alarms

Elapsed real-time alarms use the time, in milliseconds, since the device was booted. Elapsed real-time alarms aren't affected by time zones, so they work well for alarms based on the passage of time. For example, use an elapsed real-time alarm for an alarm that fires every half hour.

The AlarmManager class provides two types of elapsed real-time alarm:

- `ELAPSED_REALTIME`: Fires a `PendingIntent` based on the amount of time since the device was booted, but doesn't wake the device. The elapsed time includes any time during which the device was asleep. All repeating alarms fire when your device is next awake.

- `ELAPSED_REALTIME_WAKEUP` : Fires the `PendingIntent` after the specified length of time has elapsed since device boot, waking the device's CPU if the screen is off. Use this alarm instead of `ELAPSED_REALTIME` if your app has a time dependency, for example if it has a limited window during which to perform an operation.

Real-time clock (RTC) alarms

Real-time clock (RTC) alarms are clock-based alarms that use Coordinated Universal Time (UTC). Only choose an RTC alarm in these types of situations:

- You need your alarm to fire at a particular time of day.
- The alarm time is dependent on current locale.

Apps with clock-based alarms might not work well across locales, because they might fire at the wrong times. And if the user changes the device's time setting, it could cause unexpected behavior in your app.

The `AlarmManager` class provides two types of RTC alarm:

- `RTC` : Fires the pending intent at the specified time but doesn't wake up the device. All repeating alarms fire when your device is next awake.
- `RTC_WAKEUP` : Fires the pending intent at the specified time, waking the device's CPU if the screen is off.

Alarm best practices

Alarms affect how your app uses (or abuses) system resources. For example, imagine a popular app that syncs with a server. If the sync operation is based on clock time and every instance of the app connects to the server at the same time, the load on the server could result in delayed response times or even a "denial of service" condition.

To avoid this problem and others, follow these best practices:

- Add randomness (jitter) to network requests that trigger as a result of a repeating alarm. Here's one way to do this:
  - Schedule an exact alarm that performs any local work. "Local work" means anything that doesn't contact a server over a network or require data from that server.
  - Schedule a separate alarm that contains the network requests, and have this alarm fire after a random period of time. Usually this second alarm is set by whatever component receives the `PendingIntent` from the first alarm. (You can also set this alarm at the same time as you set the first alarm.)
- Keep your alarm frequency to a minimum.
- Don't wake up the device unnecessarily.
- Use the least precise timing possible to allow the `AlarmManager` to be the most efficient it can be. For example, when you schedule a repeating alarm, use `setInexactRepeating()` instead of `setRepeating()` . For details, see Scheduling a repeating alarm, below.
- Avoid basing your alarm on clock time and use `ELAPSED_REALTIME` for repeating alarms whenever possible. Repeating alarms that are based on a precise trigger time don't scale well.

Scheduling an alarm

The `AlarmManager` class gives you access to the Android system alarm services. `AlarmManager` lets you broadcast an `Intent` at a scheduled time, or after a specific interval.

To schedule an alarm:

1. Call `getSystemService(ALARM_SERVICE)` to get an instance of the `AlarmManager` class.
2. Use one of the `set...()` methods available in `AlarmManager` (as described below). Which method you use depends on whether the alarm is elapsed real time, or RTC.

All the `AlarmManager.set...()` methods include these two arguments:

- A `type` argument, which is how you specify the alarm type:

  - `ELAPSED_REALTIME` or `ELAPSED_REALTIME_WAKEUP`, described in Elapsed real-time alarms above.
  - `RTC` or `RTC_WAKEUP`, described in Real-time clock (RTC) alarms above.
- A `PendingIntent` object, which is how you specify which task to perform at the given time.

Scheduling a single-use alarm

To schedule a single alarm, use one of the following methods on the `AlarmManager` instance:

- `set()`: For devices running API 19+, this method schedules a single, inexactly timed alarm, meaning that the system shifts the alarm to minimize wakeups and battery use. For devices running lower API versions, this method schedules an exactly timed alarm.
- `setWindow()`: For devices running API 19+, use this method to set a window of time during which the alarm should be triggered.
- `setExact()`: For devices running API 19+, this method triggers the alarm at an exact time. Use this method only for alarms that must be delivered at an exact time, for example an alarm clock that rings at a requested time. Exact alarms reduce the OS's ability to minimize battery use, so don't use them unnecessarily.

Here's an example of using `set()` to schedule a single-use alarm:

```
alarmMgr.set(AlarmManager.ELAPSED_REALTIME,
             SystemClock.elapsedRealtime() + 1000*300,
             alarmIntent);
```

In this example:

- The `type` is `ELAPSED_REALTIME`, which means that this is an elapsed real-time alarm. If the device is idle when the alarm is sent, the alarm does not wake the device.
- The alarm is sent 5 minutes (300,000 milliseconds) after the method returns.
- `alarmIntent` is a `PendingIntent` broadcast that contains the action to perform when the alarm is sent.
  Note: For timing operations like ticks and timeouts, and events that happen more often than once a minute, it's easier and more efficient to use a Handler rather than an alarm.

Doze and App Standby

API 23+ devices sometimes enter Doze or App Standby mode to save power:

- *Doze* mode is triggered when a user leaves a device unplugged and stationary for a period of time, with the screen off. During short "maintenance windows," the system exits Doze to let apps complete deferred activities, including firing standard alarms, then returns to Doze. Doze mode ends when the user returns to their device.
- *App Standby* mode is triggered on idle apps that haven't been used recently. App Standby mode ends when the user returns to your app or plugs in the device.

To use alarms with Doze and App Standby:

- If you need an alarm that fires while a device is in Doze or App Standby mode without waiting for a maintenance window, use `setAndAllowWhileIdle()` for inexact and `setExactAndAllowWhileIdle()` for exact alarms, or set a user-visible alarm (API 21+).
- Some alarms can wait for a maintenance window, or until the device comes out of Doze or App Standby mode. For these alarms, use the standard `set()` and `setExact()` methods to optimize battery life.

Scheduling a repeating alarm

You can also use the `AlarmManager` to schedule repeating alarms, using one of the following methods:

- `setRepeating()`: Prior to Android 4.4 (API Level 19), this method creates a repeating, exactly timed alarm. On devices running API 19 and higher, `setRepeating()` behaves exactly like `setInexactRepeating()`.
- `setInexactRepeating()`: This method creates a repeating, inexact alarm that allows for batching. When you use

`setInexactRepeating()`, Android synchronizes repeating alarms from multiple apps and fires them at the same time. This reduces the total number of times the system must wake the device, thus reducing drain on the battery. As of API 19, all repeating alarms are inexact.

To decrease possible battery drain:

- Schedule repeating alarms to be as infrequent as possible.
- Use inexact timing, which allows the system to batch alarms from different apps together.

Note: while `setInexactRepeating()` is an improvement over `setRepeating()`, it can still overwhelm a server if every instance of an app hits the server around the same time. Therefore, for network requests, add some randomness to your alarms, as described in Alarm best practices.

If you really need exact repeating alarms on API 19+, set a single-use alarm with `setExact()` and set the next alarm once that alarm has triggered. This second alarm is set by whatever component receives the `PendingIntent`—usually either a service or a broadcast receiver.

Here's an example of using `setInexactRepeating()` to schedule a repeating alarm:

```
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP,
               calendar.getTimeInMillis(),
               AlarmManager.INTERVAL_FIFTEEN_MINUTES,
               alarmIntent);
```

In this example:

- The `type` is `RTC_WAKEUP`, which means that this is a clock-based alarm that wakes the device when the alarm is sent.
- The first occurrence of the alarm is sent immediately, because `calendar.getTimeInMillis()` returns the current time as UTC milliseconds.
- After the first occurrence, the alarm is sent approximately every 15 minutes.

  If the method were `setRepeating()` instead of `setInexactRepeating()`, and if the device were running an API version lower than 19, the alarm would be sent *exactly* every 15 minutes.

  Possible values for this argument are `INTERVAL_DAY`, `INTERVAL_FIFTEEN_MINUTES`, `INTERVAL_HALF_DAY`, `INTERVAL_HALF_HOUR`, `INTERVAL_HOUR`.

- `alarmIntent` is the `PendingIntent` that contains the action to perform when the alarm is sent. This intent typically comes from `IntentSender.getBroadcast()`.

Checking for an existing alarm

It's often useful to check whether an alarm is already set. For example, you may want to disable the ability to set another alarm if one already exists.

To check for an existing alarm:

1. Create a `PendingIntent` that contains the same `Intent` used to set the alarm, but this time use the `FLAG_NO_CREATE` flag.

   With `FLAG_NO_CREATE`, a `PendingIntent` is only created if one with the same `Intent` already exists. Otherwise, the request returns `null`.

2. Check whether the `PendingIntent` is `null`:
   - If it's `null`, the alarm has not yet been set.
   - If it's not `null`, the `PendingIntent` already exists, meaning that the alarm has been set.

For example, the following code returns `true` if the alarm contained in `alarmIntent` already exists:

```
boolean alarmExists =
 (PendingIntent.getBroadcast(this, 0,
       alarmIntent,
       PendingIntent.FLAG_NO_CREATE) != null);
```

Canceling an alarm

To cancel an alarm, use `cancel()` and pass in the `PendingIntent`. For example:

```
alarmManager.cancel(alarmIntent);
```

User-visible alarms ("alarm clocks")

For API 21+ devices, you can set a user-visible alarm clock by calling `setAlarmClock()`. Apps can retrieve the next user-visible alarm clock that's set to go off by calling `getNextAlarmClock()`.

Alarms clocks set with `setAlarmClock()` work even when the device or app is idle (similar to `setExactAndAllowWhileIdle()`), which gets you as close to an exact wake up call as possible.

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Alarm Manager

Learn more

- Schedule Repeating Alarms Guide
- AlarmManager reference
- Choosing an Alarm Blog Post
- Scheduling Alarms Presentation

8.3: Transferring Data Efficiently

Contents:

- Wireless radio state
- Bundling network transfers
- Prefetching
- Monitor connectivity state
- Monitor battery state
- JobScheduler
- Related practical
- Learn more

Transferring data is an essential part of most Android applications, but it can negatively affect battery life and increase data usage costs. Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain.

Users care about battery drain because they would rather use their mobile device without it connected to the charger. And users care about data usage, because every bit of data transferred can cost them money.

In this chapter, you learn how your app's networking activity affects the device's radio hardware so you can minimize the battery drain associated with network activity. You also learn how to wait for the proper conditions to accomplish resource-intensive tasks.

Wireless radio state

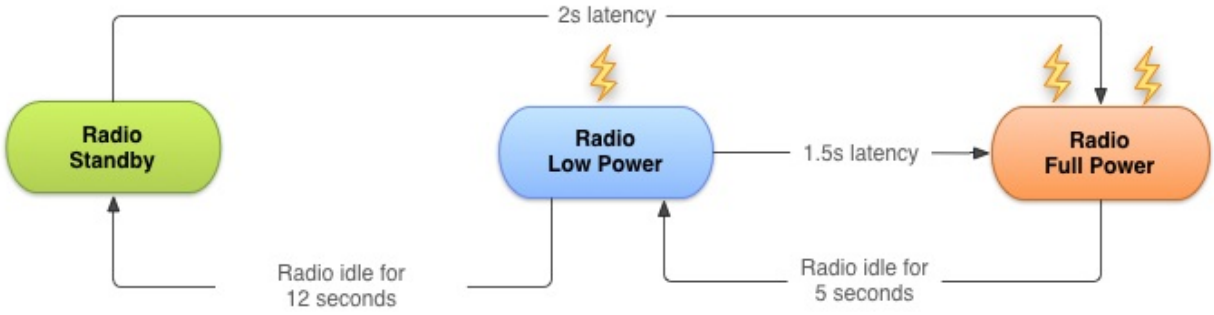
A fully active wireless radio consumes significant power. To conserve power when not in use, the radio transitions between different energy states. However, there is a trade-off between conserving power and the time it takes to power up when needed.

For a typical 3G network the radio has these three energy states:

1. **Full power:** Used when a connection is active. Allows the device to transfer data at its highest possible rate.
2. **Low power:** An intermediate state that uses about 50% less battery.
3. **Standby:** The minimal energy state, during which no network connection is active or required.

While the low and standby states use much less battery, they also introduce latency to network requests. Returning to full power from the low state takes around 1.5 seconds, while moving from standby to full can take over 2 seconds.

Android uses a state machine to determine how to transition between states. To minimize latency, the state machine waits a short time before it transitions to the lower energy states.



The radio state machine on each device, particularly the associated transition delay ("tail time") and startup latency, vary based on the wireless radio technology employed (2G, 3G, LTE, etc.) and is defined and configured by the carrier network over which the device is operating.

This chapter describes a representative state machine for a typical 3G wireless radio, based on data provided by AT&T. However, the general principles and resulting best practices are applicable for all wireless radio implementations.

As with any best practices, there are trade-offs that you need to consider for your own app development.

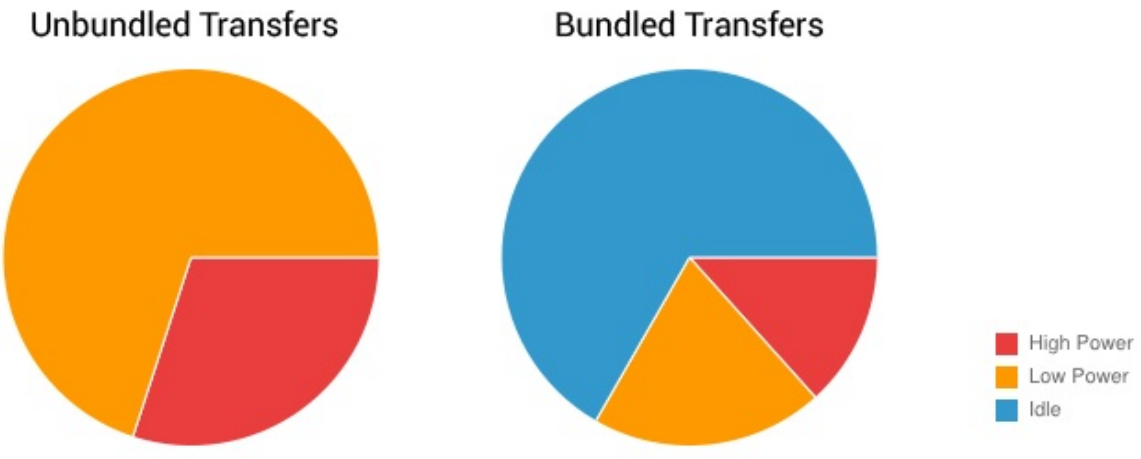
Bundling network transfers

Every time you create a new network connection, the radio transitions to the full power state. In the case of the 3G radio state machine described above, it remains at full power for the duration of your transfer, followed by 5 seconds of tail time, followed by 12 seconds at the low energy state before turning off. So, for a typical 3G device, every data transfer session causes the radio to draw power for almost 20 seconds.

What this means in practice:

- An app that transfers unbundled data for 1 second every 18 seconds keeps the wireless radio always active.
- By comparison, the same app bundling transfers for 3 seconds of every minute keeps the radio in the high power state for only 8 seconds, and in the low power state for an additional 12 seconds.

The second example allows the radio to be idle for 40 seconds out of every minute, resulting in a massive reduction in battery consumption.



It's important to bundle and queue up your data transfers. You can bundle transfers that are due to occur within a time window and make them all happen simultaneously, ensuring that the radio draws power for as little time as possible.

Prefetching

To *prefetch* data means that your app takes a guess at what content or data the user will want next, and fetches it ahead of time. For example, when the user looks at the first part of an article, a good guess is to prefetch the next part. Or, if a user is watching a video, fetching the next minutes of the video is also a good guess.

Prefetching data is an effective way to reduce the number of independent data transfer sessions. Prefetching allows you to download all the data you are likely to need for a given time period in a single burst, over a single connection, at full capacity. This reduces the number of radio activations required to download the data. As a result, you not only conserve battery life, but also improve latency for the user, lower the required bandwidth, and reduce download times.

Prefetching has trade-offs. If you download too much or the wrong data, you might increase battery drain. And if you download at the wrong time, users may end up waiting. Optimizing prefetching data is an advanced topic not covered in this course, but the following guidelines cover common situations.

How aggressively you prefetch depends on the size of the data being downloaded and the likelihood of it being used. As a rough guide, based on the state machine described above, for data that has a 50% chance of being used within the current user session, you can typically prefetch for around 6 seconds (approximately 1-2 Mb) before the potential cost of

downloading unused data matches the potential savings of not downloading that data to begin with.

Generally speaking, it's good practice to prefetch data such that you only need to initiate another download every 2 to 5 minutes, and on the order of 1 to 5 megabytes.

Following this principle, large downloads—such as video files—should be downloaded in chunks at regular intervals (every 2 to 5 minutes), effectively prefetching only the video data likely to be viewed in the next few minutes.

Prefetching example

Many news apps attempt to reduce bandwidth by downloading headlines only after a category has been selected, full articles only when the user wants to read them, and thumbnails just as they scroll into view.

Using this approach, the radio is forced to remain active for the majority of a news-reading session as users scroll headlines, change categories, and read articles. Not only that, but the constant switching between energy states results in significant latency when switching categories or reading articles.

Here's a better approach:

1. Prefetch a reasonable amount of data at startup, beginning with the first set of news headlines and thumbnails. This ensures a quick startup time.
2. Continue with the remaining headlines, the remaining thumbnails, and the article text for each article from the first set of headlines.

Monitor connectivity state

Devices can network using different types of hardware:

- *Wireless radios* use varying amounts of battery depending on technology, and higher bandwidth consumes more energy. Higher bandwidth means you can prefetch more aggressively, downloading more data during the same amount of time. However, perhaps less intuitively, because the tail-time battery cost is relatively higher, it is also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.
- *WiFi radio* uses significantly less battery than wireless and offers greater bandwidth.

Perform data transfers when connected over Wi-Fi whenever possible.

You can use the `ConnectivityManager` to determine the active wireless radio and modify your prefetching routines depending on network type:

```
ConnectivityManager cm =
   (ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);
TelephonyManager tm =
    (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
    case (ConnectivityManager.TYPE_WIFI):
        PrefetchCacheSize = MAX_PREFETCH_CACHE; break;
    case (ConnectivityManager.TYPE_MOBILE): {
        switch (tm.getNetworkType()) {
           case (TelephonyManager.NETWORK_TYPE_LTE |
                  TelephonyManager.NETWORK_TYPE_HSPAP):
                PrefetchCacheSize *= 4;
                break;
            case (TelephonyManager.NETWORK_TYPE_EDGE |
                  TelephonyManager.NETWORK_TYPE_GPRS):
               PrefetchCacheSize /= 2;
               break;
            default: break;
        }
        break;
      }
  default: break;
  }
```

The system sends out broadcast intents when the connectivity state changes, so you can listen for these changes using a `BroadcastReceiver`.

Monitor battery state

To minimize battery drain, monitor the state of your battery and wait for specific conditions before initiating a battery-intensive operation.

The `BatteryManager` broadcasts all battery and charging details in a broadcast `Intent` that includes the charging status.

To check the current battery status, examine the broadcast intent:

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

If you want to react to changes in the battery charging state, use a `BroadcastReceiver` registered for the battery status actions:

```
<receiver android:name=".PowerConnectionReceiver">
  <intent-filter>
    <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
    <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
  </intent-filter>
</receiver>
```

Broadcast intents are also delivered when the battery level changes in a significant way:

```
"android.intent.action.BATTERY_LOW"
"android.intent.action.BATTERY_OKAY"
```

JobScheduler

Constantly monitoring the connectivity and battery status of the device can be a challenge, and it requires using components such as broadcast receivers, which can consume system resources even when your app isn't running. Because transferring data efficiently is such a common task, the Android SDK provides a class that makes this much easier: `JobScheduler`.

Introduced in API level 21, `JobScheduler` allows you to schedule a task around specific conditions (rather than a specific time as with `AlarmManager`).

`JobScheduler` has three components:

1. `JobInfo` uses the builder pattern to set the conditions for the task.
2. `JobService` is a wrapper around the `Service` class where the task is actually completed.
3. `JobScheduler` schedules and cancels tasks.
   Note: `JobScheduler` is only available from API 21+. There is no backwards compatible version for prior API releases. If your app targets devices with earlier API levels, you might find the FirebaseJobDispatcher a useful alternative.

1. JobInfo

Set the job conditions by constructing a `JobInfo` object using the `JobInfo.Builder` class. The `JobInfo.Builder` class is instantiated from a constructor that takes two arguments: a job ID (which can be used to cancel the job), and the `ComponentName` of the `JobService` that contains the task. Your `JobInfo.Builder` must set at least one, non-default condition for the job. For example:

```
JobScheduler scheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
ComponentName serviceName = new ComponentName(getPackageName(),
NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName);
builder.setRequiredNetworkType(NETWORK_TYPE_UNMETERED);
JobInfo jobInfo = builder.build();
```

Note: See the related practical for a complete example.
The JobInfo.Builder class has many `set()` methods that allow you to determine the conditions of the task. Below is a list of available constraints with their respective `set()` methods and class constants:

- **Backoff/Retry policy:** Determines when how the task should be rescheduled if it fails. Set this condition using the `setBackoffCriteria()` method, which takes two arguments: the initial time to wait after the task fails, and the backoff strategy. The backoff strategy argument can be one of two constants: `BACKOFF_POLICY_LINEAR` or `BACKOFF_POLICY_EXPONENTIAL`. This defaults to {30 seconds, Exponential}.
- **Minimum Latency:** The minimum amount of time to wait before completing the task. Set this condition using the `setMinimumLatency()` method, which takes a single argument: the amount of time to wait in milliseconds.
- **Override Deadline:** The maximum time to wait before running the task, even if other conditions aren't met. Set this condition using the `setOverrideDeadline()` method, which is the maximum time to wait in milliseconds.
- **Periodic:** Repeats the task after a certain amount of time. Set this condition using the `setPeriodic()` method, passing in the repetition interval. This condition is mutually exclusive with the minimum latency and override deadline conditions: setting `setPeriodic()` with one of them results in an error.
- **Persisted:** Sets whether the job is persisted across system reboots. For this condition to work, your app must hold the `RECEIVE_BOOT_COMPLETED` permission. Set this condition using the `setPersisted()` method, passing in a boolean that indicates whether or not to persist the task.
- **Required Network Type:** The kind of network type your job needs. If the network isn't necessary, you don't need to call this function, because the default is `NETWORK_TYPE_NONE`. Set this condition using the `setRequiredNetworkType()`

method, passing in one of the following constants: `NETWORK_TYPE_NONE`, `NETWORK_TYPE_ANY`, `NETWORK_TYPE_NOT_ROAMING`, `NETWORK_TYPE_UNMETERED`.

- **Required Charging State:** Whether or not the device needs to be plugged in to run this job. Set this condition using the `setRequiresCharging()` method, passing in a boolean. The default is `false`.
- **Requires Device Idle:** Whether or not the device needs to be in idle mode to run this job. "Idle mode" means that the device isn't in use and hasn't been for some time, as loosely defined by the system. When the device is in idle mode, it's a good time to perform resource-heavy jobs. Set this condition using the `setRequiresDeviceIdle()` method, passing in a boolean. The default is `false`.

2. JobService

Once the conditions for a task are met, the framework launches a subclass of `JobService`, which is where you implement the task itself. The `JobService` runs on the UI thread, so you need to offload blocking operations to a worker thread.

Declare the `JobService` subclass in the Android Manifest, and include the `BIND_JOB_SERVICE` permission:

```
<service android:name="MyJobService"
             android:permission="android.permission.BIND_JOB_SERVICE" />
```

In your subclass of `JobService`, override two methods, `onStartJob()` and `onStopJob()`.

onStartJob()

The system calls `onStartJob()` and automatically passes in a `JobParameters` object, which the system creates with information about your job. If your task contains long-running operations, offload the work onto a separate thread. The `onStartJob()` method returns a boolean: `true` if your task has been offloaded to a separate thread (meaning it might not be completed yet) and `false` if there is no more work to be done.

Use the `jobFinished()` method from any thread to tell the system that your task is complete. This method takes two parameters: the `JobParameters` object that contains information about the task, and a boolean that indicates whether the task needs to be rescheduled, according to the defined backoff policy.

onStopJob()

The system calls `onStopJob()` if it determines that you must stop execution of your job even before you've call `jobFinished()`. This happens if the requirements that you specified when you scheduled the job are no longer met.

Examples:

- If you request WiFi with `setRequiredNetworkType()` but the user turns off WiFi while while your job is executing, the system calls `onStopJob()`.
- If you specify `setRequiresDeviceIdle()` but the user starts interacting with the device while your job is executing, the system calls `onStopJob()`.

You're responsible for how your app behaves when it receives `onStopJob()`, so don't ignore it. This method returns a boolean, indicating whether you'd like to reschedule the job based on the defined backoff policy, or drop the task.

3. JobScheduler

The final part of scheduling a task is to use the `JobScheduler` class to schedule the job. To obtain an instance of this class, call `getSystemService(JOB_SCHEDULER_SERVICE)`. Then schedule a job using the `schedule()` method, passing in the `JobInfo` object you created with the `JobInfo.Builder`. For example:

```
mScheduler.schedule(myJobInfo);
```

The framework is intelligent about when you receive callbacks, and it attempts to batch and defer them as much as possible. Typically, if you don't specify a deadline on your job, the system can run it at any time, depending on the current state of the `JobScheduler` object's internal queue; however, it might be deferred as long as until the next time the device is connected to a power source.

To cancel a job, call `cancel()`, passing in the job ID from the `JobInfo.Builder` object, or call `cancelAll()`. For example:

```
mScheduler.cancelAll();
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- JobScheduler

Learn more

- Transferring Data Without Draining the Battery Guide
- Optimizing Downloads for Efficient Network Access Guide
- Modifying your Download Patterns Based on the Connectivity Type Guide
- JobScheduler Reference
- JobService Reference
- JobInfo Reference
- JobInfo.Builder Reference
- JobParameters Reference
- Presentation on Scheduling Tasks
- What are the different networks and speeds?

9.0: Storing Data

Contents:

- Shared preferences
- Files
- SQLite database
- Other storage options
- Network connection
- Backing up app data
- Firebase
- Learn more

Android provides several options for you to save persistent application data. The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications (and the user) and how much space your data requires.

Your data storage options are the following:

- Shared preferences—Store private primitive data in key-value pairs. This will be covered in the next chapter.
- Internal storage—Store private data on the device memory.
- External storage—Store public data on the shared external storage.
- SQLite databases—Store structured data in a private database.
- Network connection—Store data on the web with your own network server.
- Cloud Backup—Backing up app and user data in the cloud.
- Content providers—Store data privately and make them available publicly. This will be covered in the after-next chapter.
- Firebase realtime database—Store and sync data with a NoSQL cloud database. Data is synced across all clients in real time, and remains available when your app goes offline.

Shared preferences

Using shared preferences is a way to read and write key-value pairs of information persistently to and from a file.

Note: By default these key-value pairs are neither shared nor preferences, so do not confuse them with the Preference APIs.
Shared Preferences is covered in its ">own chapter.

Files

Android uses a file system that's similar to disk-based file systems on other platforms such as Linux. File-based operations should be familiar to anyone who has used use Linux file I/O or the java.io package.

All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage).

Today, some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not. The following lists summarize the facts about each storage space.

Internal storage	External storage
Always available.	Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable.	World-readable. Any app can read.
When the user uninstalls your app, the system removes all your app's files from internal storage.	When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Internal storage

You don't need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.

You can create files in two different directories:

- Permanent storage: `getFilesDir()`
- Temporary storage: `getCacheDir()`. Recommended for small, temporary files totalling less than 1MB. Note that the system may delete temporary files if it runs low on memory.

To create a new file in one of these directories, you can use the File() constructor, passing the File provided by one of the above methods that specifies your internal storage directory. For example:

```
File file = new File(context.getFilesDir(), filename);
```

Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory. For example, here's how to write some text to a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Or, if you need to cache some files, instead use createTempFile(). For example, the following method extracts the filename from a URL and creates a file with that name in your app's internal cache directory:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

External storage

Use external storage for files that should be permanently stored, even if your app is uninstalled, and be available freely to other users and apps, such as pictures, drawings, or documents made by your app.

Some private files that are of no value to other apps can also be stored on external storage. Such files might be additional downloaded app resources, or temporary media files. Make sure you delete those when your app is uninstalled.

Obtain permissions for external storage

To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file. This implicitly includes permission to read.

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

If your app needs to read the external storage (but not write to it), then you will need to declare the READ_EXTERNAL_STORAGE permission.

```
<manifest ...>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    ...
</manifest>
```

Always check whether external storage is mounted

Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it. You can query the external storage state by calling getExternalStorageState(). If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files. For example, the following methods are useful to determine the storage availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Public and private external storage

External storage is very specifically structured and used by the Android system. There are public directories, and private directories specific to your app. Each of these file trees has directories identified by system constants.

For example, any files that you store into the public ringtone directory DIRECTORY_RINGTONES are available to all other ringtone apps.

On the other hand, any files you store in a private ringtone directory DIRECTORY_RINGTONES can, by default, only be seen by your app and are deleted along with your app.

See list of public directories for the full listing.

Getting file descriptors

To access a public external storage directory, get a path and create a file calling getExternalStoragePublicDirectory().

```
File path = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES);
File file = new File(path, "DemoPicture.jpg");
```

To access a private external storage directory, get a path and create a file calling getExternalFilesDir().

```
File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
```

Querying storage space

If you know ahead of time how much data you're saving, you can find out whether sufficient space is available without causing an IOException by calling getFreeSpace() or getTotalSpace(). These methods provide the current available space and the total space in the storage volume, respectively.

You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an IOException if one occurs. You may need to do this if you don't know exactly how much space you need.

Deleting files

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call delete()on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the Context to locate and delete a file by calling deleteFile():

```
myContext.deleteFile(fileName);
```

As a good citizen, you should also regularly delete cached files that you created with getCacheDir().

Interacting with files summary

Once you have the file descriptors, use standard java.io file operators or streams to interact with the files. This is not Android-specific and not covered here.

SQLite database

Saving data to a database is ideal for repeating or structured data, such as contact information. Android provides an SQL-like database for this purpose.

The following chapters and practical will teach you in depth how to use SQLite databases with your Android apps:

- SQLite Primer
- Introduction to SQLite Databases
- SQLite Data Storage Practical
- Searching an SQLite Database Practical

Other storage options

Android provides additional storage options that are beyond the scope of this introductory course. If you'd like to explore them, see the resources below.

Network connection

You can use the network (when it's available) to store and retrieve data on your own web-based services. To do network operations, use classes in the following packages:

- java.net.*
- android.net.*

Backing up app data

Users often invest significant time and effort creating data and setting preferences within apps. Preserving that data for users if they replace a broken device or upgrade to a new one is an important part of ensuring a great user experience.

Auto backup for Android 6.0 (API level 23) and higher

For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically backup app data to the cloud. The system performs this automatic backup for nearly all app data by default, and does so without your having to write any additional app code.

When a user installs your app on a new device, or reinstalls your app on one (for example, after a factory reset), the system automatically restores the app data from the cloud. The automatic backup feature preserves the data your app creates on a user device by uploading it to the user's Google Drive account and encrypting it. There is no charge to you or the user for

data storage, and the saved data does not count towards the user's personal Google Drive quota. Each app can store up to 25MB. Once its backed-up data reaches 25MB, the app no longer sends data to the cloud. If the system performs a data restore, it uses the last data snapshot that the app had sent to the cloud.

Automatic backups occur when the following conditions are met:

- The device is idle.
- The device is charging.
- The device is connected to a Wi-Fi network.
- At least 24 hours have elapsed since the last backup.

You can customize and configure auto backup for your app. See Configuring Auto Backup for Apps.

Backup for Android 5.1 (API level 22) and lower

For users with previous versions of Android, you need to use the Backup API to implement data backup. In summary, this requires you to:

1. Register for the Android Backup Service to get a Backup Service Key.
2. Configure your Manifest to use the Backup Service.
3. Create a backup agent by extending the BackupAgentHelper class.
4. Request backups when data has changed.

More information and sample code:

- Using the Backup API
- Data Backup

Firebase

Firebase is a mobile platform that helps you develop apps, grow your user base, and earn more money. Firebase is made up of complementary features that you can mix-and-match to fit your needs.

Some features are Analytics, Cloud Messaging, Notifications, and the Test Lab.

For data management, Firebase offers a Realtime Database.

- Store and sync data with a NoSQL cloud database.
- Connected apps share data
- Hosted in the cloud
- Data is stored as JSON
- Data is synchronized in real time to every connected client
- Data remains available when your app goes offline

See the Firebase home for more information.

Learn more

Files

- Saving Files
- getExternalFilesDir() documentation and code samples
- getExternalStoragePublicDirectory() documentation and code samples
- java.io.File class
- Oracle's Java I/O Tutorial

Backup

- Configuring Auto Backup for Apps
- Using the Backup API
- Data Backup

Shared Preferencss

- Saving Key-Value Sets
- Using Shared Preferences Guide
- Shared Preferences reference

Firebase

- Firebase home
- Firebase Realtime Database
- Add Firebase to Your Android Project

9.1: Shared Preferences

Contents:

- Shared preferences vs. saved instance state
- Creating a shared preferences file
- Saving shared preferences
- Restoring shared preferences
- Clearing shared preferences
- Listening for preference changes
- Related practical
- Learn more

Shared preferences allow you to read and write small amounts of primitive data as key/value pairs to a file on the device storage. The SharedPreference class provides APIs for getting a handle to a preference file and for reading, writing, and managing this data. The shared preferences file itself is managed by the framework and accessible to (shared with) all the components of your app. That data is not shared with or accessible to any other apps.

For managing large amounts of data, use a SQLite database or other suitable storage option, which will be discussed in a later chapter.

Shared preferences vs. saved instance state

In a previous chapter you learned about preserving state using saved instance states. Here is a comparison between the two.

Shared Preferences	Saved instance state
Persists across user sessions, even if your app is killed and restarted, or the device is rebooted.	Preserves state data across activity instances in the same user session.
Data that should be remembered across sessions, such as a user's preferred settings or their game score.	Data that should not be remembered across sessions, such as the currently selected tab, or any current state of an activity.
Small number of key/value pairs.	Small number of key/value pairs.
Data is private to the application.	Data is private to the application.
Common use is to store user preferences.	Common use is to recreate state after the device has been rotated.

Note: The SharedPreference APIs are also different from the Preference APIs. The Preference APIs can be used to build user interface for a settings page, and they do use shared preferences for their underlying implementation. See Settings for more information on settings and the Preference APIs.

Creating a shared preferences file

You need only one shared preferences file for your app, and it is customarily named with the package name of your app. This makes its name unique and easily associated with your app.

You create the shared preferences file in the onCreate() method of your main activity and store it in a member variable.

```
private String sharedPrefFile = "com.example.android.hellosharedprefs";
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

The mode argument is required, because older versions of Android had other modes that allowed you to create a world-readable or world-writable shared preferences file. These modes were deprecated in API 17, and are now **strongly discouraged** for security reasons. If you need to share data with other apps, use a service or a content provider.

Saving shared preferences

You save preferences in the onPause() state of the activity lifecycle using tge SharedPreferences.Editor interface.

1. Get a SharedPreferences.Editor. The editor takes care of all the file operations for you. When two editors are modifying preferences at the same time, the last one to call apply wins.
2. Add key/value pairs to the editor using the put method appropriate for the data type. The put methods will overwrite previously existing values of an existing key.
3. Call apply() to write out your changes. The apply() method saves the preferences asynchronously, off of the UI thread. The shared preferences editor also has a commit() method to synchronously save the preferences. The commit() method is discouraged as it can block other operations. As SharedPreferences instances are singletons within a process, it's safe to replace any instance of commit() with apply() if you were already ignoring the return value.

   You don't need to worry about Android component lifecycles and their interaction with apply() writing to disk. The framework makes sure in-flight disk writes from apply() complete before switching states.

```
@Override
protected void onPause() {
   super.onPause();
   SharedPreferences.Editor preferencesEditor = mPreferences.edit();
   preferencesEditor.putInt("count", mCount);
   preferencesEditor.putInt("color", mCurrentColor);
   preferencesEditor.apply();
}
```

Restoring shared preferences

You restore shared preferences in the onCreate() method of your activity. The get() methods take two arguments—one for the key and one for the default value if the key cannot be found. Using the default argument, you don't have to test whether the preference exists in the file.

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
if (savedInstanceState != null) {
    mCount = mPreferences.getInt("count", 1);
    mShowCount.setText(String.format("%s", mCount));

    mCurrentColor = mPreferences.getInt("color", mCurrentColor);
    mShowCount.setBackgroundColor(mCurrentColor);
} else { … }
```

Clearing shared preferences

To clear all the values in the shared preferences file, call the clear() method on the shared preferences editor and apply the changes.

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.putInt("number", 42);
preferencesEditor.clear();
preferencesEditor.apply();
```

You can combine calls to put and clear. However, when applying the preferences, the clear is always done first, regardless of whether you called clear before or after the put methods on this editor.

Listening for preference changes

There are several reasons you might want to be notified as soon as the user changes one of the preferences. In order to receive a callback when a change happens to any one of the preferences, implement the SharedPreference.OnSharedPreferenceChangeListener interface and register the listener for the SharedPreferences object by calling registerOnSharedPreferenceChangeListener().

The interface has only one callback method, onSharedPreferenceChanged(), and you can implement the interface as a part of your activity.

```
public class SettingsActivity extends PreferenceActivity
                              implements OnSharedPreferenceChangeListener {
    public static final String KEY_PREF_SYNC_CONN = "pref_syncConnectionType";
    ...

    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
        String key) {
        if (key.equals(KEY_PREF_SYNC_CONN)) {
            Preference connectionPref = findPreference(key);
            // Set summary to be the user-description for the selected value
            connectionPref.setSummary(sharedPreferences.getString(key, ""));
        }
    }
}
```

In this example, the method checks whether the changed setting is for a known preference key. It calls findPreference() to get the Preference object that was changed so it can modify the item's summary to be a description of the user's selection.

For proper lifecycle management in the activity, register and unregister your SharedPreferences.OnSharedPreferenceChangeListener during the onResume() and onPause() callbacks, respectively:

```
@Override
protected void onResume() {
    super.onResume();
    getPreferenceScreen().getSharedPreferences()
            .registerOnSharedPreferenceChangeListener(this);
}

@Override
protected void onPause() {
    super.onPause();
    getPreferenceScreen().getSharedPreferences()
            .unregisterOnSharedPreferenceChangeListener(this);
}
```

Hold a reference to the listener

When you call registerOnSharedPreferenceChangeListener(), the preference manager does not currently store a reference to the listener. You must hold onto a reference to the listener, or it will be susceptible to garbage collection. Keep a reference to the listener in the instance data of an object that will exist as long as you need the listener.

```
SharedPreferences.OnSharedPreferenceChangeListener listener =
    new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
    // listener implementation
  }
};
prefs.registerOnSharedPreferenceChangeListener(listener);
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Shared Preferences

Learn more

- Saving Data
- Storage Options
- Saving Key-Value Sets
- SharedPreferences
- SharedPreferences.Editor

Stackoverflow

- How to use SharedPreferences in Android to store, fetch and edit values
- onSavedInstanceState vs. SharedPreferences

9.2: App Settings

Contents:

- Determining appropriate setting controls
- Providing navigation to Settings
- The Settings UI
- Displaying the settings
- Setting the default values for settings
- Reading the settings values
- Listening for a setting change
- Using the Settings Activity template
- Related practical
- Learn more

This chapter describes app settings that let users indicate their preferences for how an app or service should behave.

Determining appropriate setting controls

Apps often include settings that allow users to modify app features and behaviors. For example, some apps allow users to specify whether notifications are enabled, or how often the application syncs data with the cloud.

The controls that belong in the app's settings should capture user preferences that affect most users or provide critical support to a minority of users. For example, notification settings affect all users, while a currency setting for a foreign market provides critical support for the users in that market.

Settings are usually accessed infrequently, because once users change a setting, they rarely need to go back and change it again. If the control or preference you are providing the user is something that needs to be frequently accessed, consider moving it to the app bar's options menu, or to a side navigation menu such as a navigation drawer.

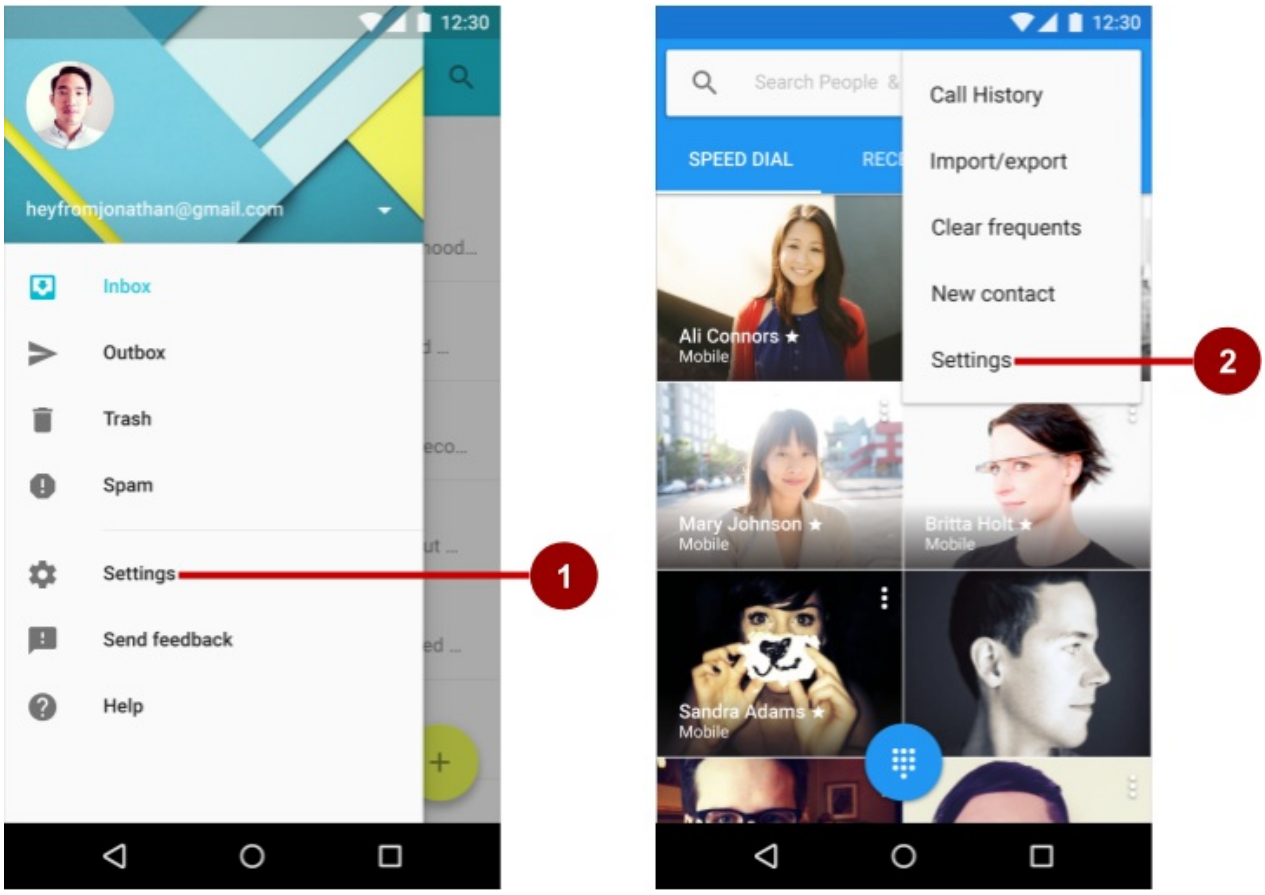
Set defaults for your setting controls that are familiar to users and make the app experience better. The initial default value for a setting should:

- Represent the value most users would choose, such as **All contacts** for "Contacts to display" in the Contacts app.
- Use less battery power. For example, in the Android Settings app, Bluetooth is set to off until the user turns it on.
- Pose the least risk to security and data loss. For example, the default setting for the Gmail app's default action is to archive rather than delete messages.
- Interrupt only when important. For example, the default setting for when calls and notifications arrive is to interrupt only when important.

Tip: If the setting contains information about the app, such as a version number or licensing information, move these to a separately-accessed Help screen.

Providing navigation to Settings

Users should be able to navigate to app settings by tapping **Settings**, which should be located in side navigation, such as a navigation drawer, as shown on the left side of the figure below, or in the options menu in the app bar, as shown on the right side of the figure below.



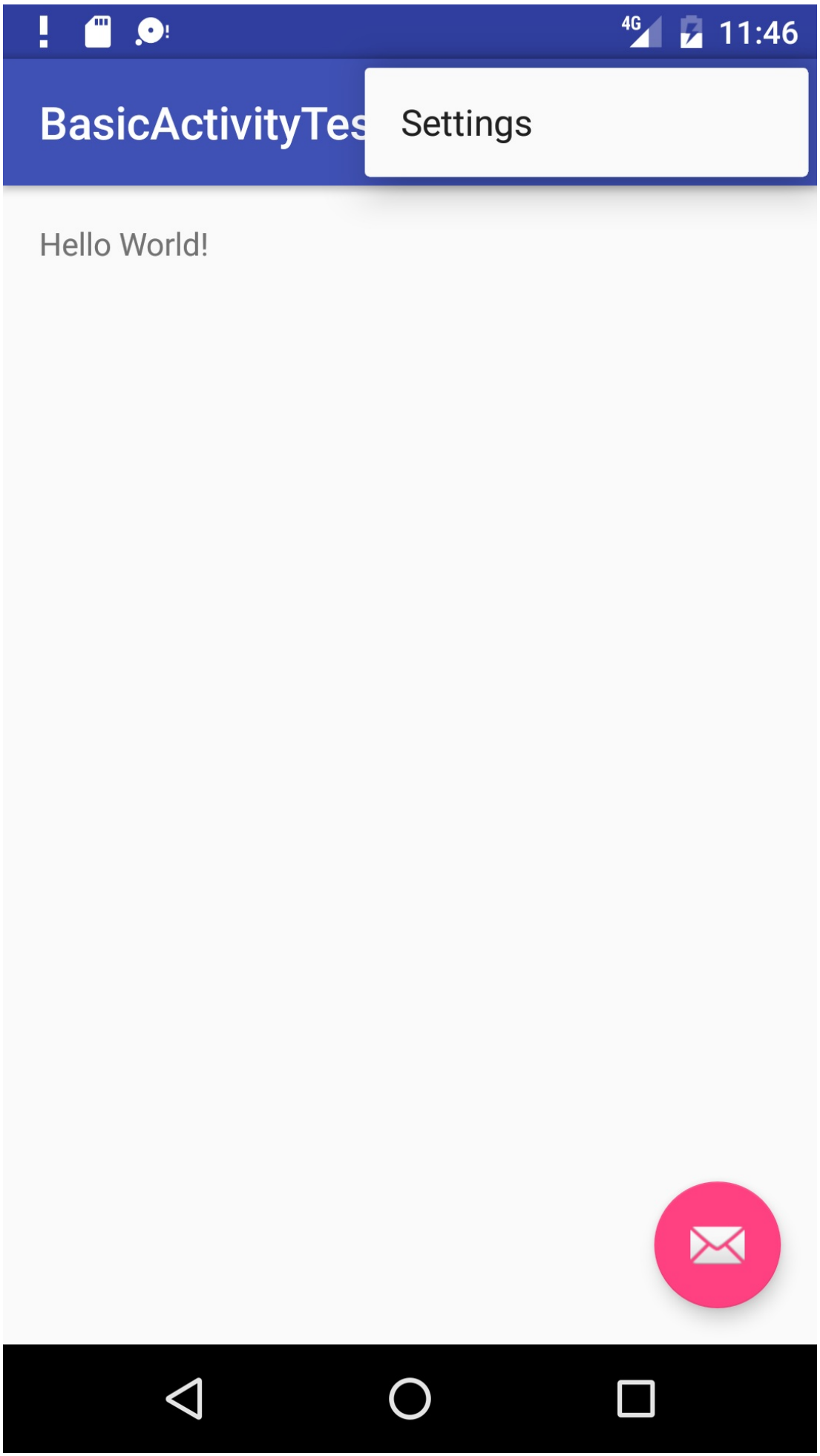
In the figure above:

1. Settings in side navigation (a navigation drawer)
2. Settings in the options menu of the app bar

Follow these design guidelines for navigating to settings:

- If your app offers side navigation such as a navigation drawer, include **Settings** below all other items (except **Help** and **Send Feedback**).
- If your app doesn't offer side navigation, place **Settings** in the app bar menu's options menu below all other items (except **Help** and **Send Feedback**).
  Note: Use the word **Settings** in the app's navigation to access the settings. Do not use synonyms such as "Options" or "Preferences."

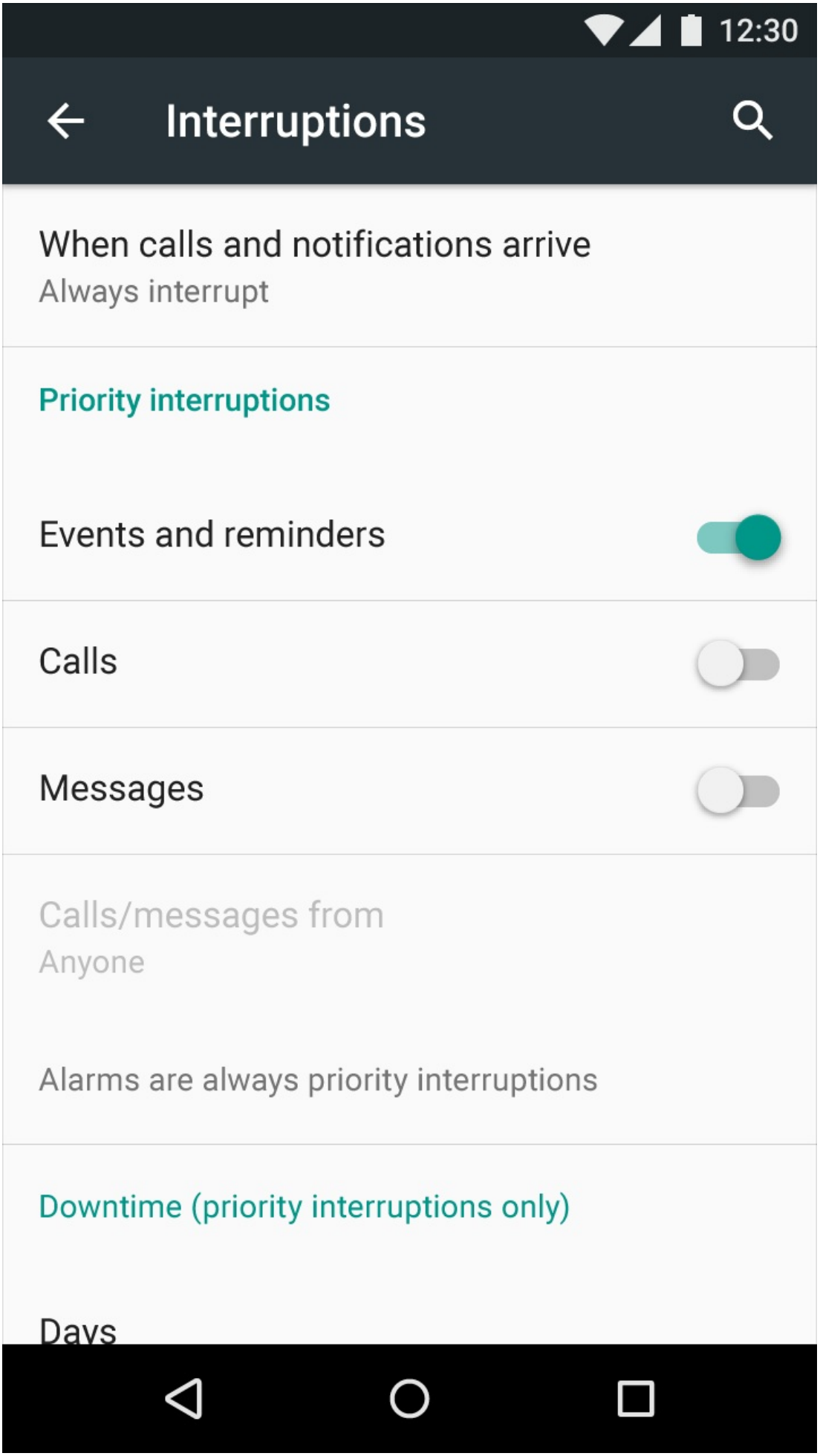
Tip: Android Studio provides a shortcut for setting up an options menu with **Settings**. If you start an Android Studio project for a smartphone or tablet using the Basic Activity template, the new app includes **Settings** as shown below:


11:46
BasicActivityTes
Settings
Hello World!

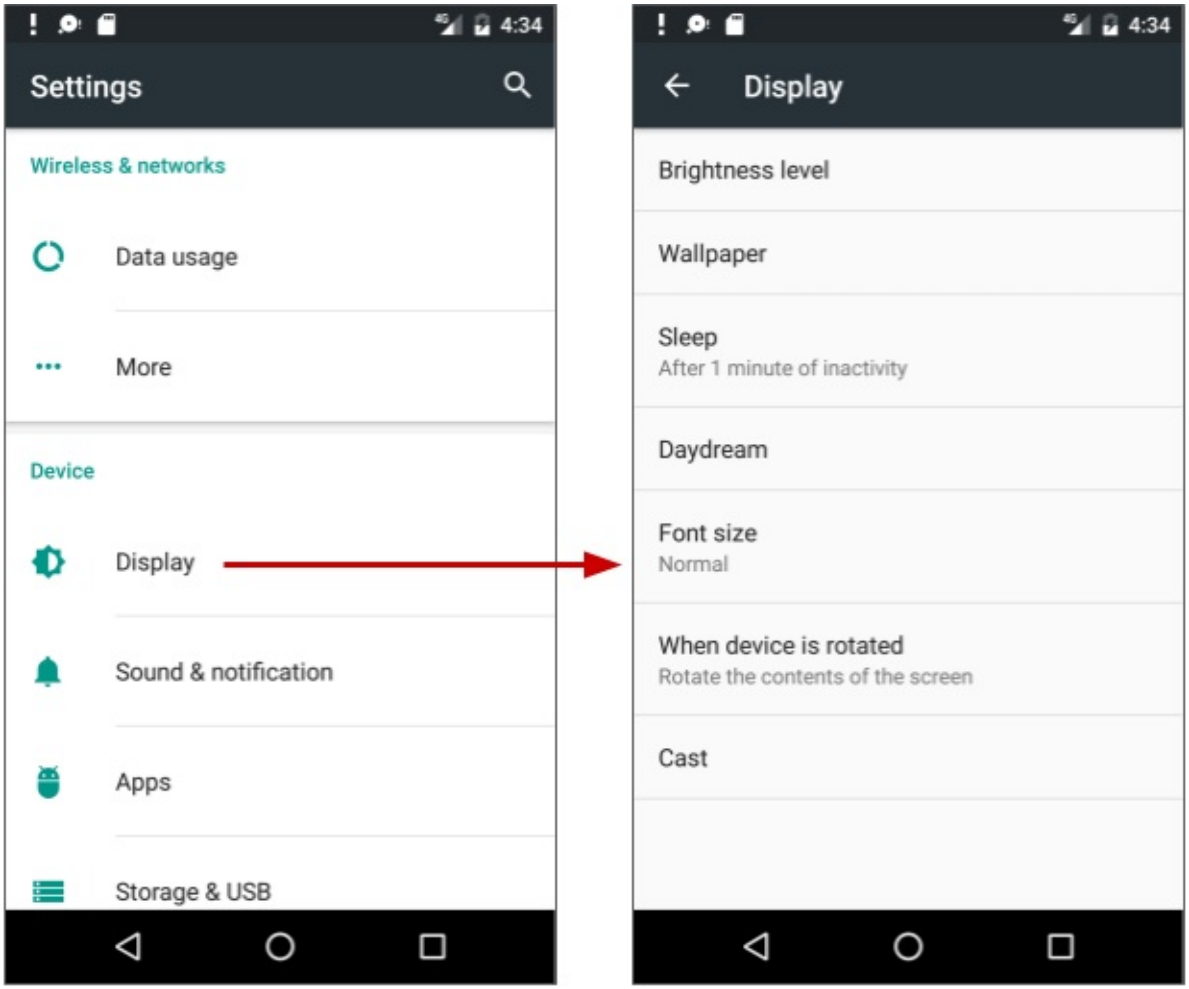
The Settings UI

Settings should be well-organized, predictable, and contain a manageable number of options. A user should be able to quickly understand all available settings and their current values. Follow these design guidelines:

- **7 or fewer settings**: Arrange them according to priority with the most important ones at the top.
- **7-15 settings**: Group related settings under section dividers. For example, in the figure below, "Priority interruptions" and "Downtime (priority interruptions only)" are section dividers.


12:30
Interruptions
When calls and notifications arrive
Always interrupt
Priority interruptions
Events and reminders
Calls
Messages
Calls/messages from
Anyone
Alarms are always priority interruptions
Downtime (priority interruptions only)
Days

- **16 or more settings**: Group related settings into separate sub-screens. Use headings, such as **Display** on the main Settings screen (as shown on the left side of the figure below) to enable users to navigate to the display settings (shown on the right side of the figure below):



Building the settings

Build an app's settings using various subclasses of the Preference class rather than using View objects. This class provides the View to be displayed for each setting, and associates with it a SharedPreferences interface to store/retrieve the preference data.

Each Preference appears as an item in a list. Direct subclasses provide containers for layouts involving multiple settings. For example:

- PreferenceGroup: Represents a group of settings (Preference objects).
- PreferenceCategory: Provides a disabled title above a group as a section divider.
- PreferenceScreen: Represents a top-level Preference that is the root of a Preference hierarchy. Use a PreferenceScreen in a layout at the top of each screen of settings.

For example, to provide dividers with headings between groups of settings (as shown in the previous figure for 7-15 settings), place each group of Preference objects inside a PreferenceCategory. To use separate screens for groups, place each group of Preference objects inside a PreferenceScreen.

Other Preference subclasses for settings provide the appropriate UI for users to change the setting. For example:

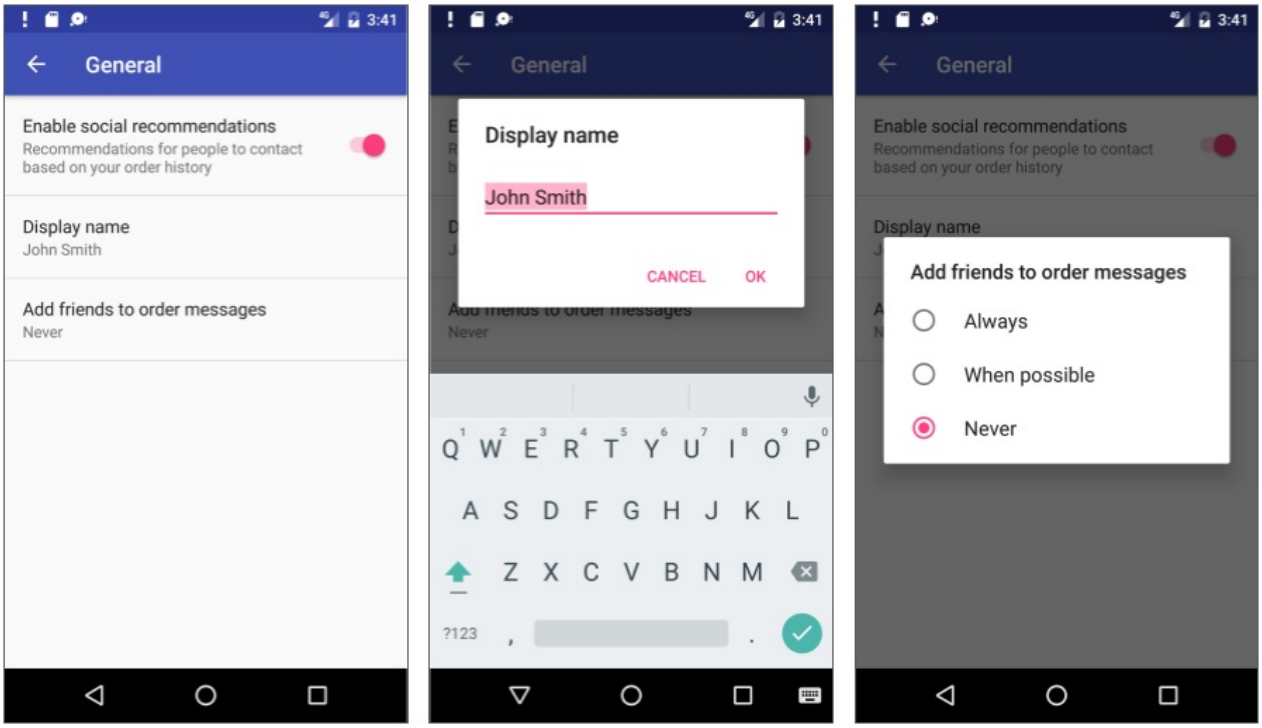
- CheckBoxPreference: Creates a list item that shows a checkbox for a setting that is either enabled or disabled. The saved value is a boolean (`true` if it's checked).
- ListPreference: Creates an item that opens a dialog with a list of radio buttons.
- SwitchPreference: Creates a two-state toggleable option (such as on/off or true/false).

- EditTextPreference: Creates an item that opens a dialog with an EditText widget. The saved value is a String.
- RingtonePreference: Lets the user to choose a ringtone from those available on the device.

Define your list of settings in XML, which provides an easy-to-read structure that's simple to update. Each Preference subclass can be declared with an XML element that matches the class name, such as `<CheckBoxPreference>`.

XML attributes for settings

The following example from the Settings Activity template defines a screen with three settings as shown in the figure below: a toggle switch (at the top of the screen on the left side), a text entry field (center), and a list of radio buttons (right):

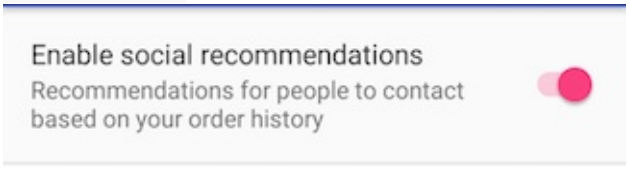


The root of the settings hierarchy is a `PreferenceScreen` layout:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
. . .
</PreferenceScreen>
```

Inside this layout are three settings:

- `SwitchPreference`: Show a toggle switch to disable or enable an option.

  The setting has the following attributes:

  - `android:defaultValue`: The option is enabled (set to `true`) by default.
  - `android:summary`: The text summary appears underneath the setting. For some settings, the summary should change to show whether the option is enabled or disabled.
  - `android:title`: The title of the setting. For a `SwitchPreference`, the title appears to the left of the toggle switch.
  - android:key: The key to use for storing the setting value. Each setting (Preference object) has a corresponding key-value pair that the system uses to save the setting in a default SharedPreferences file for your app's settings.

```
<SwitchPreference
     android:defaultValue="true"
     android:key="example_switch"
     android:summary="@string/pref_description_social_recommendations"
     android:title="@string/pref_title_social_recommendations" />
```

- `EditTextPreference` : Show a text field for the user to enter text.

  - Use EditText attributes such as `android:capitalize` and `android:maxLines` to define the text field's appearance and input control.
  - The default setting is the `pref_default_display_name` string resource.

```
<EditTextPreference
   android:capitalize="words"
   android:defaultValue="@string/pref_default_display_name"
   android:inputType="textCapWords"
   android:key="example_text"
   android:maxLines="1"
   android:selectAllOnFocus="true"
   android:singleLine="true"
   android:title="@string/pref_title_display_name" />
```

- `ListPreference` : Show a dialog with radio buttons for the user to make one choice.

  - The default value is set to `-1` for no choice.
  - The text for the radio buttons (Always, When possible, and Never) are defined in the `pref_example_list_titles` array and specified by the `android:entries` attribute.
  - The values for the radio button choices are defined in the `pref_example_list_values` array and specified by the `android:entryValues` attribute.
  - The radio buttons are displayed in a dialog, which usually have positive (**OK** or **Accept**) and negative (**Cancel**) buttons. However, a settings dialog doesn't need these buttons, because the user can touch outside the dialog to dismiss it. To hide these buttons, set the `android:positiveButtonText` and `android:negativeButtonText` attributes to `"@null"` .

```
<ListPreference
   android:defaultValue="-1"
   android:entries="@array/pref_example_list_titles"
   android:entryValues="@array/pref_example_list_values"
   android:key="example_list"
   android:negativeButtonText="@null"
   android:positiveButtonText="@null"
   android:title="@string/pref_title_add_friends_to_messages" />
```

Save the XML file in the **res/xml/** directory. Although you can name the file anything you want, it's traditionally named **preferences.xml**.

If you are using the support v7 appcompat library and extending the Settings Activity with AppCompatActivity and the fragment with PreferenceFragmentCompat, as shown in the next section, change the setting's XML attribute to use the support v7 appcompat library version. For example, for a SwitchPreference setting, change `<SwitchPreference` in the code to:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
   <android.support.v7.preference.SwitchPreferenceCompat
   ... />
</PreferenceScreen>
```

Displaying the settings

Use a specialized Activity or Fragment subclass to display a list of settings.

- For an app that supports Android 3.0 and newer versions, the best practice for settings is to use a Settings Activity and a fragment for each preference XML file:
  - Add a Settings Activity class that extends Activity and hosts a fragment that extends PreferenceFragment.
  - To remain compatible with the v7 appcompat library, extend the Settings Activity with AppCompatActivity, extend the fragment with PreferenceFragmentCompat.
- If your app must support versions of Android older than 3.0 (API level 10 and lower), build a special settings activity as an extension of the PreferenceActivity class.

Fragments like PreferenceFragment provide a more flexible architecture for your app, compared to using activities alone. A fragment is like a modular section of an activity—it has its own lifecycle and receives its own input events, and you can add or remove a fragment while the activity is running. Use PreferenceFragment to control the display of your settings instead of PreferenceActivity whenever possible.

However, to create a two-pane layout for large screens when you have multiple groups of settings, you can use an activity that extends PreferenceActivity and also use PreferenceFragment to display each list of settings. You will see this pattern with the Settings Activity template as described later in this chapter in "Using the Settings Activity template".

The following examples show you how to remain compatible with the v7 appcompat library by extending the Settings Activity with AppCompatActivity, and extending the fragment with PreferenceFragmentCompat. To use this library and the `PreferenceFragmentCompat` version of PreferenceFragment, you must also add the android.support:preference-v7 library to the **build.gradle (Module: app)** file's `dependencies` section:

```
dependencies {
   ...
   compile 'com.android.support:preference-v7:25.0.1'
}
```

You also need to add the following `preferenceTheme` declaration to the `AppTheme` in the **styles.xml** file:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
   ...
   <item name="preferenceTheme">@style/PreferenceThemeOverlay</item>
</style>
```

Using a PreferenceFragment

The following shows how to use a PreferenceFragment to display a list of settings, and how to add a PreferenceFragment to an activity for settings. To remain compatible with the v7 appcompat library, extend the Settings Activity with AppCompatActivity, and extend the fragment with PreferenceFragmentCompat for each preferences XML file.

Replace the automatically generated `onCreate()` method with the onCreatePreferences() method to load a preferences file with `setPreferencesFromResource()`:

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreatePreferences(Bundle savedInstanceState,
                                    String rootKey) {
        setPreferencesFromResource(R.xml.preferences, rootKey);
    }
}
```

As shown in the code above, you associate an XML layout of settings with the fragment during the `onCreatePreferences()` callback by calling setPreferencesFromResource() with two arguments:

- `R.xml.` and the name of the XML file (`preferences`).
- the `rootKey` to identify the preference root in `PreferenceScreen` .

  ```
  setPreferencesFromResource(R.xml.preferences, rootKey);
  ```

You can then create an Activity for settings (named `SettingsActivity`) that extends AppCompatActivity, and add the settings fragment to it:

```
public class SettingsActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Display the fragment as the main content.
        getSupportFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
        ...
    }
}
```

The above code is the typical pattern used to add a fragment to an activity so that the fragment appears as the main content of the activity. You use:

- `getFragmentManager()` if the class extends `Activity` and the fragment extends `PreferenceFragment` .
- `getSupportFragmentManager()` if the class extends `AppCompatActivity` and the fragment extends `PreferenceFragmentCompat` .

For more information about fragments, see Fragment.

To set Up navigation for the settings activity, be sure to declare the Settings Activity's parent to be MainActivity in the AndroidManifest.xml file.

Calling the settings activity

If you implement the options menu with the **Settings** item, use the following intent to call the Settings activity from with the `onOptionsItemSelected()` method when the user taps **Settings** (using `action_settings` for the **Settings** menu resource id):

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   int id = item.getItemId();
   // ... Handle other options menu items
   if (id == R.id.action_settings) {
      Intent intent = new Intent(this, SettingsActivity.class);
      startActivity(intent);
      return true;
      }
   return super.onOptionsItemSelected(item);
}
```

If you implement a navigation drawer with the **Settings** item, use the following intent to call the Settings activity from with the `onNavigationItemSelected()` method when the user taps **Settings** (using `action_settings` for the **Settings** menu resource id):

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
   int id = item.getItemId();
   if (id == R.id.action_settings) {
      Intent intent = new Intent(this, SettingsActivity.class);
      startActivity(intent);
   } else if ...
   // ... Handle other navigation drawer items
   return true;
}
```

Setting the default values for settings

When the user changes a setting, the system saves the changes to a SharedPreferences file. As you learned in another lesson, shared preferences allow you to read and write small amounts of primitive data as key/value pairs to a file on the device storage.

The app must initialize the SharedPreferences file with default values for each setting when the user first opens the app. Follow these steps:

1. Be sure to specify a default value for each setting in your XML file using the `android:defaultValue` attribute:

   ```
   ...
   <SwitchPreference
           android:defaultValue="true"
           ... />
   ...
   ```

2. From the onCreate() method in the app's main activity—and in any other activity through which the user may enter your app for the first time—call setDefaultValues():

   ```
   ...
     PreferenceManager.setDefaultValues(this,
                               R.xml.preferences, false);
   ...
   ```

Step 2 ensures that the app is properly initialized with default settings. The `setDefaultValues()` method takes three arguments:

- The app context, such as `this`.
- The resource ID (`preferences`) for the settings layout XML file which includes the default values set by Step 1 above.

- A boolean indicating whether the default values should be set more than once. When `false`, the system sets the default values only if this method has never been called in the past (or the KEY_HAS_SET_DEFAULT_VALUES in the default value SharedPreferences file is false). As long as you set this third argument to `false`, you can safely call this method every time your activity starts without overriding the user's saved settings values by resetting them to the default values. However, if you set it to `true`, the method will override any previous values with the defaults.

Reading the settings values

Each Preference you add has a corresponding key-value pair that the system uses to save the setting in a default SharedPreferences file for your app's settings. When the user changes a setting, the system updates the corresponding value in the SharedPreferences file for you. The only time you should directly interact with the associated SharedPreferences file is when you need to read the value in order to determine your app's behavior based on the user's setting.

All of an app's preferences are saved by default to a file that is accessible from anywhere within the app by calling the static method PreferenceManager.getDefaultSharedPreferences(). This method takes the context and returns the SharedPreferences object containing all the key-value pairs that are associated with the Preference objects.

For example, the following code snippet shows how you can read one of the preference values from the main activity's `onCreate()` method:

```
...
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(this);
Boolean switchPref = sharedPref
    .getBoolean("example_switch", false);
...
```

The above code snippet uses `PreferenceManager.getDefaultSharedPreferences(this)` to get the settings as a SharedPreferences object (`sharedPref`).

It then uses `getBoolean()` to get the boolean value of the preference that uses the key `"example_switch"`. If there is no value for the key, the `getBoolean()` method sets the value to `false`.

Listening for a setting change

There are several reasons why you might want to set up a listener for a specific setting:

- If a change to the value of a setting also requires changing the summary of the setting, you can listen for the change, and then change the summary with the new setting value.
- If the setting requires several more options, you may want to listen for the change and immediately respond by displaying the options.
- If the setting makes another setting obsolete or inappropriate, you may want to listen for the change and immediately respond by disabling the other setting.

To listen to a setting, use the Preference.OnPreferenceChangeListener interface, which includes the onPreferenceChange() method that returns the new value of the setting.

In the following example, the listener retrieves the new value after the setting is changed, and changes the summary of the setting (which appears below the setting in the UI) to show the new value. Follow these steps:

1. Use a shared preferences file, as described in a previous chapter, to store the value of the preference (setting). Declare the following variables in the SettingsFragment class definition:

```
public class SettingsFragment extends PreferenceFragment {
   private SharedPreferences mPreferences;
   private String sharedPrefFile = "com.example.android.settingstest";
   ...
}
```

2. Add the following to the `onCreate()` method of SettingsFragment to get the preference defined by the key `example_switch`, and to set the initial text (the string resource `option_on`) for the summary:

```
@Override
public void onCreate(Bundle savedInstanceState) {
   ...
   mPreferences =
            this.getActivity()
            .getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
   Preference preference = this.findPreference("example_switch");
   preference.setSummary(mPreferences.getString("summary",
                                getString(R.string.option_on)));
   ...
}
```

3. Add the following code to `onCreate()` after the code in the previous step:

```
...
preference.setOnPreferenceChangeListener(new
                          Preference.OnPreferenceChangeListener() {
   @Override
   public boolean onPreferenceChange(Preference preference,
                          Object newValue) {
      if ((Boolean) newValue == true) {
         preference.setSummary(R.string.option_on);
         SharedPreferences.Editor preferencesEditor =
                                          mPreferences.edit();
         preferencesEditor.putString("summary",
                          getString(R.string.option_on)).apply();
      } else {
         preference.setSummary(R.string.option_off);
         SharedPreferences.Editor preferencesEditor =
                                          mPreferences.edit();
         preferencesEditor.putString("summary",
                          getString(R.string.option_off)).apply();
      }
      return true;
   }
});
...
```

The code does the following:

i. Listens to a change in the switch setting using onPreferenceChange(), and returns `true`:

```
@Override
public boolean onPreferenceChange(Preference preference,
                  Object newValue) {
 ...
 return true;
}
```

ii. Determines the new boolean value (`newValue`) for the setting after the change (`true` or `false`):

```
if ((Boolean) newValue == true) {
 ...
} else {
 ...
}
```

iii. Edits the Shared Preferences file (as described in a previous practical) using SharedPreferences.Editor:

```
...
preference.setSummary(R.string.option_on);
SharedPreferences.Editor preferencesEditor =
                                    mPreferences.edit();
...
```

iv. Puts the new value as a string in the summary using putString() and applies the change using apply():

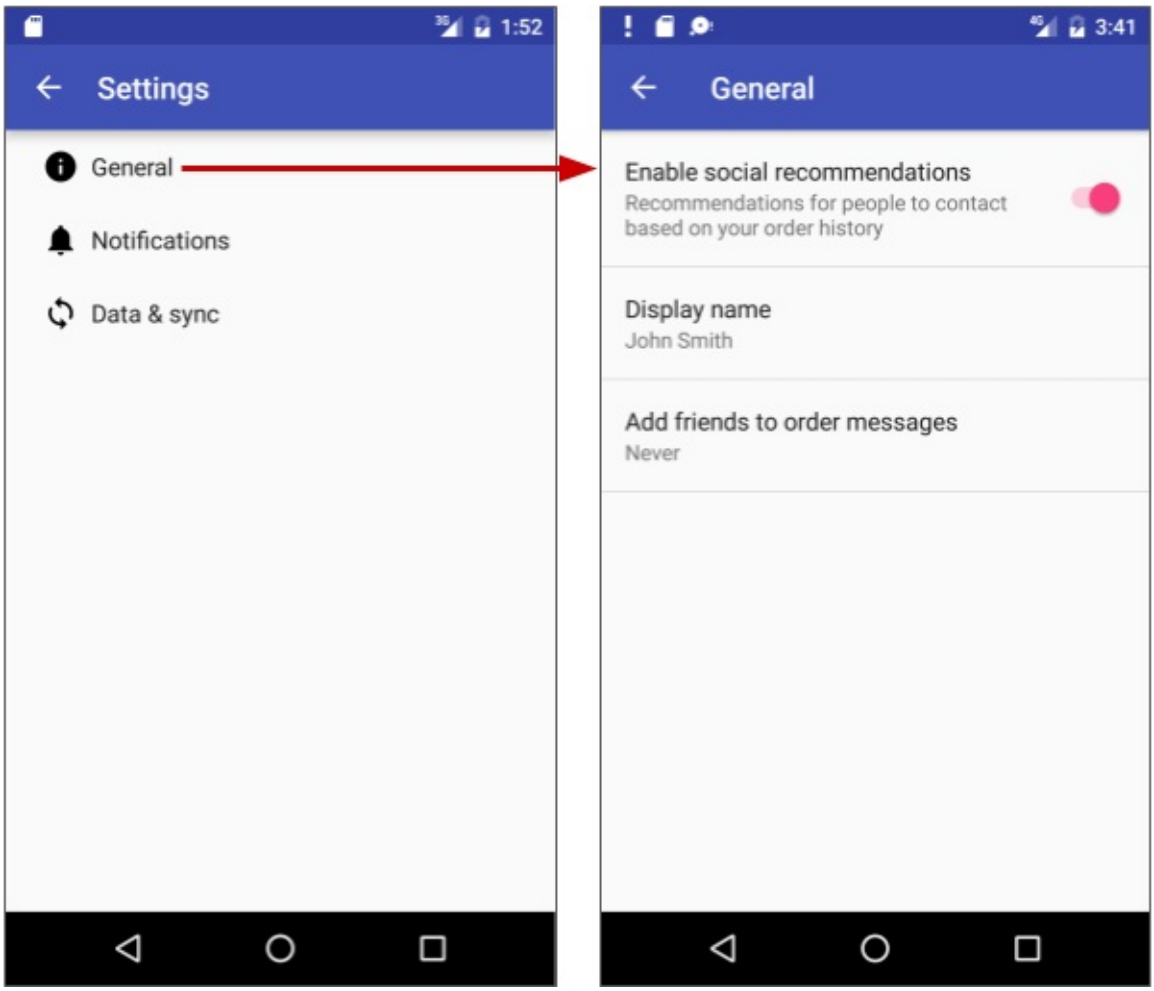
```
...
preferencesEditor.putString("summary",
                    getString(R.string.option_on)).apply();
...
```

Using the Settings Activity template

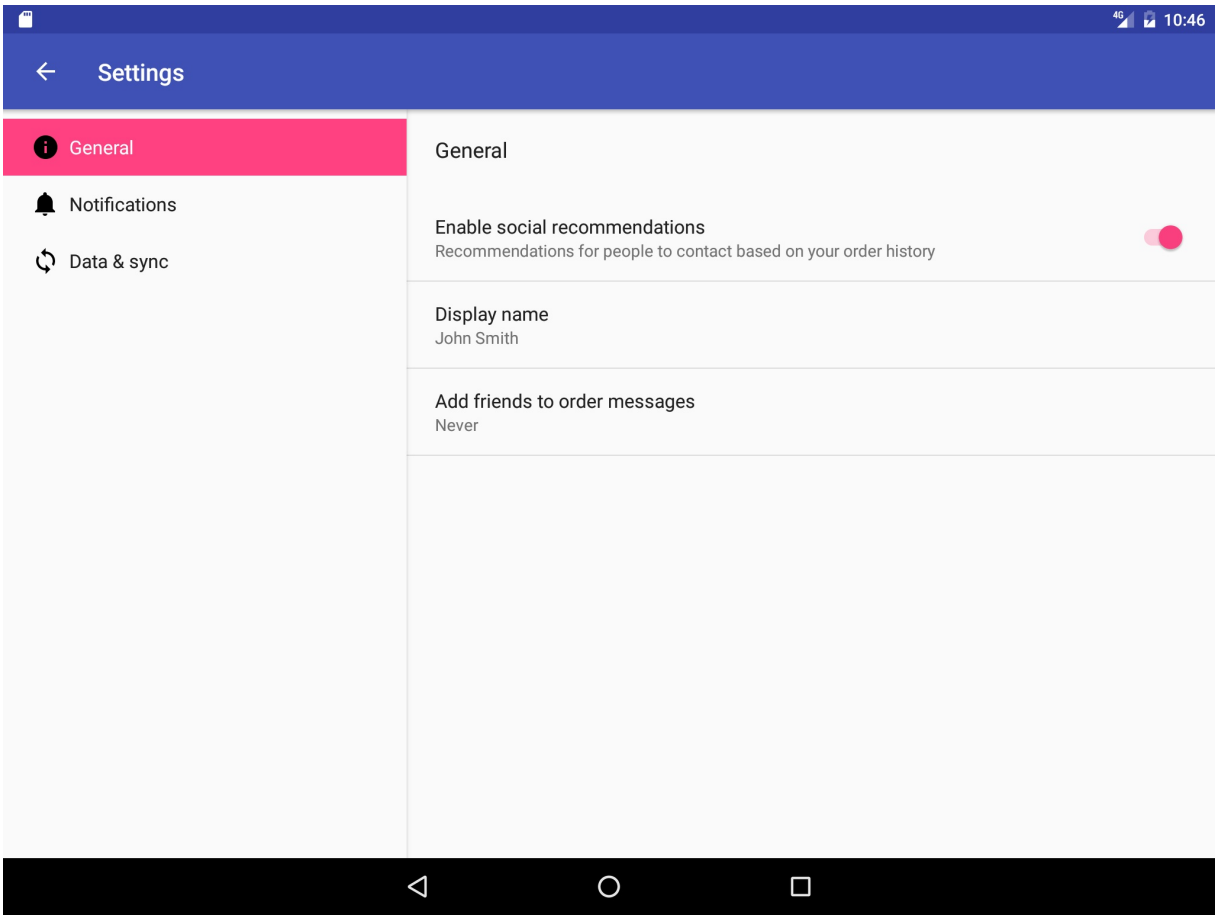
If you need to build several sub-screens of settings and you want to take advantage of tablet-sized screens as well as maintain compatibility with older versions of Android for tablets, Android Studio provides a shortcut: the Settings Activity template.

The Settings Activity template is pre-populated with settings you can customize for an app, and provides a different layout for smartphones and tablets:

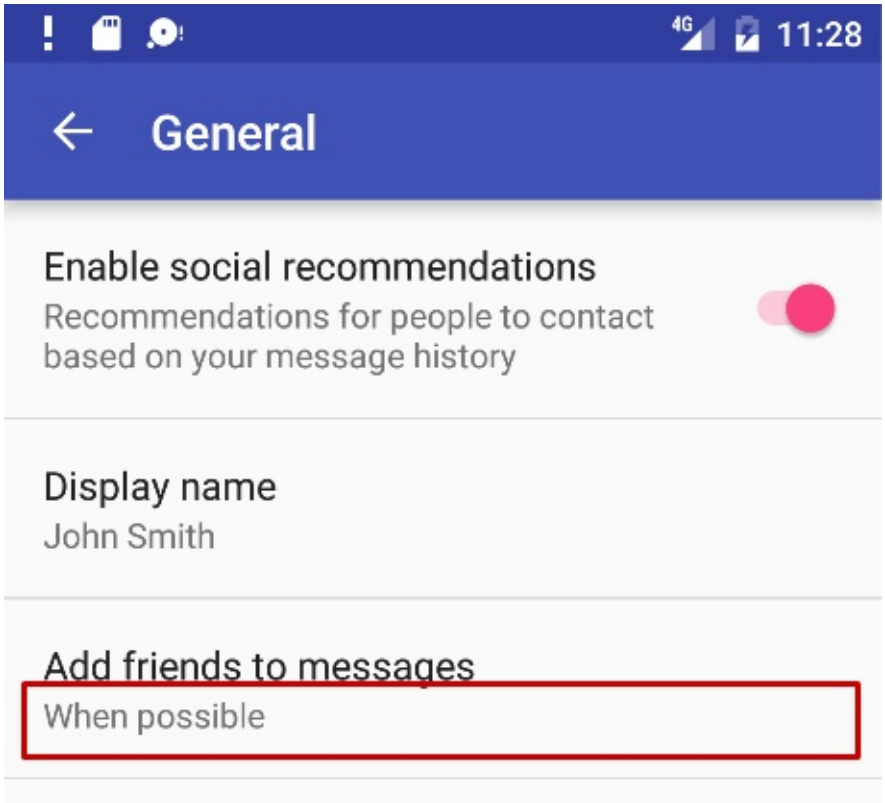
- *Smartphones*: A main Settings screen with a header link for each group of settings, such as General for general settings, as shown below.



- *Tablets*: A master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



The Settings Activity template also provides the function of listening to a settings change, and changing the summary to reflect the settings change. For example, if you change the "Add friends to messages" setting (the choices are **Always**, **When possible**, or **Never**), the choice you make appears in the summary underneath the setting:



In general, you need not change the Settings Activity template code in order to customize the activity for the settings you want in your app. You can customize the settings titles, summaries, possible values, and default values without changing the template code, and even add more settings to the groups that are provided. To customize the settings, edit the string and string array resources in the strings.xml file and the layout attributes for each setting in the files in the **xml** directory.

You use the Settings Activity template code as-is. To make it work for your app, add code to the Main Activity to set the default settings values, and to *read* and *use* the settings values, as shown later in this chapter.

Including the Settings Activity template in your project

To include the Settings Activity template in your app project in Android Studio, follow these steps:

1. Choose **New > Activity > Settings Activity**.
2. In the dialog that appears, accept the Activity Name (**SettingsActivity** is the suggested name) and the Title (**Settings**).
3. Click the three dots at the end of the Hierarchical Parent field and choose the parent activity (usually **MainActivity**), so that Up navigation in the Settings Activity returns the user to the MainActivity. Choosing the parent activity automatically updates the AndroidManifest.xml file to support Up navigation.

The Settings Activity template creates the XML files in the **res > xml** directory, which you can add to or customize for the settings you want:

- pref_data_sync.xml: PreferenceScreen layout for "Data & Sync" settings.
- pref_general.xml: PreferenceScreen layout for "General" settings.
- pref_headers.xml: Layout of headers for the Settings main screen.
- pref_notification.xml: PreferenceScreen layout for "Notifications" settings.

  The above XML layouts use various subclasses of the Preference class rather than View objects, and direct subclasses provide containers for layouts involving multiple settings. For example, PreferenceScreen represents a top-level Preference that is the root of a Preference hierarchy. The above files use PreferenceScreen at the top of each screen of settings. Other Preference subclasses for settings provide the appropriate UI for users to change the setting. For example:

  - CheckBoxPreference: A checkbox for a setting that is either enabled or disabled.
  - ListPreference: A dialog with a list of radio buttons.
  - SwitchPreference: A two-state toggleable option (such as on/off or true/false).
  - EditTextPreference: A dialog with an EditText widget.
  - RingtonePreference: A dialog with ringtones on the device.

The Settings Activity template also provides the following:

- String resources in the strings.xml file in the **res > values** directory, which you can customize for the settings you want.

  All strings used in the Settings Activity, such as the titles for settings, string arrays for lists, and descriptions for settings, are defined as string resources at the end of this file. They are marked by comments such as `<!-- Strings related to Settings -->` and `<!-- Example General settings -->`.

  Tip: You can edit these strings to customize the settings you need for your app.

- SettingsActivity in the **java > com.example.android.*projectname*** directory, which you can use as is.

  The activity that displays the settings. `SettingsActivity` extends `AppCompatPreferenceActivity` for maintaining compatibility with older versions of Android.

- AppCompatPreferenceActivity in the **java > com.example.android.*projectname*** directory, which you use as is.

  This activity is a helper class that SettingsActivity uses to maintain backwards compatibility with previous versions of Android.

Using preference headers

The Settings Activity template shows preference headers on the main screen that separate the settings into categories (**General**, **Notifications**, and **Data & sync**). The user taps a heading to access the settings under that heading. On larger tablet displays (see previous figure), the headers appear in the left pane and the settings for each header appears in the right pane.

To implement the headers, the template provides the pref_headers.xml file:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
   <header
      android:fragment=
           "com.example.android.droidcafe.SettingsActivity$GeneralPreferenceFragment"
      android:icon="@drawable/ic_info_black_24dp"
      android:title="@string/pref_header_general" />

   <header
      android:fragment=
      "com.example.android.droidcafe.SettingsActivity$NotificationPreferenceFragment"
      android:icon="@drawable/ic_notifications_black_24dp"
      android:title="@string/pref_header_notifications" />

   <header
      android:fragment=
          "com.example.android.droidcafe.SettingsActivity$DataSyncPreferenceFragment"
      android:icon="@drawable/ic_sync_black_24dp"
      android:title="@string/pref_header_data_sync" />
</preference-headers>
```

The XML headers file lists each preferences category and declares the fragment that contains the corresponding preferences.

To display the headers, the template uses the following `onBuildHeaders()` method:

```
@Override
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.pref_headers, target);
}
```

The above code snippet uses the `loadHeadersFromResource()` method of the PreferenceActivity class to load the headers from the XML resource (pref_headers.xml). The TargetApi annotation tells Android Studio's Lint code scanning tool that the class or method is targeting a particular API level regardless of what is specified as the min SDK level in manifest. Lint would otherwise produce errors and warnings when using new functionality that is not available in the target API level.

Using PreferenceActivity with fragments

The Settings Activity template provides an activity (SettingsActivity) that extends PreferenceActivity to create a two-pane layout to support large screens, and *also* includes fragments within the activity to display lists of settings. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as smartphones.

The following shows how to use an activity that extends `PreferenceActivity` to host one or more fragments (PreferenceFragment) that display app settings. The activity can host multiple fragments, such as `GeneralPreferenceFragment` and `NotificationPreferenceFragment`, and each fragment definition uses `addPreferencesFromResource` to load the settings from the XML preferences file:

```
public class SettingsActivity extends AppCompatPreferenceActivity {
   ...
   public static class GeneralPreferenceFragment extends
                                            PreferenceFragment {
   @Override
   public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.pref_general);

   ...
   }
   public static class NotificationPreferenceFragment extends
                                            PreferenceFragment {
   ...
}
```

Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Adding Settings to an App

Learn more

- Android Studio documentation:
  - Android Studio User Guide
- Android API Guide, "Develop" section:
  - Settings (coding)
  - Preference class
  - PreferenceFragment
  - Fragment
  - SharedPreferences
  - Saving Key-Value Sets
- Material Design Specification:
  - Settings (design)

10.0: SQLite Primer

Contents:

- SQL databases
- SQLite
- Example table
- Transactions
- Query language
- Queries for Android SQLite
- Cursors
- Learn more

This course assumes that you are familiar with databases in general, SQL databases in particular, and the SQL language used to interact with them. This chapter is a refresher and quick reference only.

SQL databases

- Store data in tables of rows and columns.
- The intersection of a row and column is called a field.
- Fields contain data, references to other fields, or references to other tables.
- Rows are identified by unique IDs.
- Columns are identified by names that are unique per table.

Think of it as a spreadsheet with rows, columns, and cells, where cells can contain data, references to other cells, and links to other sheets.

SQLite

SQLite is a software library that implements SQL database engine that is:

- self-contained (requires no other components)
- serverless (requires no server backend)
- zero-configuration (does not need to be configured for your application)
- transactional (changes within a single transaction in SQLite either occur completely or not at all)

SQLite is the most widely deployed database engine in the world. The source code for SQLite is in the public domain.

For details of the SQLite database, see the SQLite website.

Example table

SQLite stores data in tables.

Assume the following:

- A database DATABASE_NAME
- A table WORD_LIST_TABLE
- Columns for _id, word, and description

After inserting the words "alpha" and "beta", where alpha has two definitions, the table might look like this:

DATABASE_NAME

WORD_LIST_TABLE		
_id	**word**	**definition**
1	"alpha"	"first letter"
2	"beta"	"second letter"
3	"alpha"	"particle"

You can find what's in a specific row using the _id, or you can retrieve rows by formulating queries that select rows from the table be specifying constraints. You use the SQL query language discussed below to create queries.

Transactions

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

All changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by

- a program crash,
- an operating system crash, or
- a power failure.

Examples of transactions:

- Transferring money from a savings account to a checking account.
- Entering a term and definition into dictionary.
- Committing a changelist to the master branch.

ACID

- **Atomicity.** Either all of its data modifications are performed, or none of them are performed.
- **Consistency.** When completed, a transaction must leave all data in a consistent state.
- **Isolation.** Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state.
- **Durability.** After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

More on transactions.

Query language

You use a special SQL query language to interact with the database. Queries can be very complex, but the basic operations are

- inserting rows
- deleting rows
- updating values in rows
- retrieving rows that meet given criteria

On Android, the database object provides convenient methods for inserting, deleting, and updating the database. You only need to understand SQL for retrieving data.

Full description of the query language.

Query structure

A SQL query is highly structured and contains the following basic parts:

- SELECT word, description FROM WORD_LIST_TABLE WHERE word="alpha"

Generic version of sample query:

- SELECT columns FROM table WHERE column="value"

Parts:

- SELECT columns—select the columns to return. Use * to return all columns.
- FROM table—specify the table from which to get results.
- WHERE—keyword for conditions that have to be met.
- column="value"—the condition that has to be met.
  - common operators: =, LIKE, <, >
- AND, OR—connect multiple conditions with logic operators.
- ORDER BY—omit for default order, or specify ASC for ascending, DESC for descending.
- LIMIT is a very useful keyword if you want to only get a limited number of results.

Sample queries

1	SELECT * FROM WORD_LIST_TABLE	Get the whole table.
2	SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > 2	Returns `[["alpha", "particle"]]`
3	SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%"	Return the id of the word alpha with the substring "art" in the definition. `[["3"]]`
4	SELECT * FROM WORD_LIST_TABLE ORDER BY word DESC LIMIT 1	Sort in reverse and get the first item. This gives you the last item per sort order. Sorting is by the first column, in this case, the _id. `[["3", "alpha", "particle"]]`
5	SELECT * FROM WORD_LIST_TABLE LIMIT 2,1	Returns 1 item starting at position 2. Position counting starts at 1 (not zero!). Returns `[["2", "beta", "second letter"]]`

You can practice creating and querying databases at this Fiddle website and HeadFirst Labs.

Queries for Android SQLite

You can send queries to the SQLite database of the Android system as raw queries or as parameters.

- rawQuery(String sql, String[] selectionArgs) runs the provided SQL and returns a Cursor of the result set.

The following table shows how the first two queries from above would look as raw queries.

1	String query = "SELECT * FROM WORD_LIST_TABLE"; rawQuery(query, null);
2	query = "SELECT word, definition FROM WORD_LIST_TABLE WHERE _id> ? "; String[] selectionArgs = new String[]{"2"} rawQuery(query, selectionArgs) ;

- query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit) queries the given table, returning a Cursor over the result set.

Here's a query showing how to fill in the arguments:

```
SELECT * FROM WORD_LIST_TABLE
WHERE word="alpha"
ORDER BY word ASC
LIMIT 2,1;
```

Returns:

```
[["alpha", "particle"]]
String table = "WORD_LIST_TABLE"
String[] columns = new String[]{"*"};
String selection = "word = ?"
String[] selectionArgs = new String[]{"alpha"};
String groupBy = null;
String having = null;
String orderBy = "word ASC"
String limit = "2,1"

query(table, columns, selection, selectionArgs, groupBy, having, orderBy, limit);
```

Note that in real code, you wouldn't create variables for null values. See the SQLiteDatabase documentation for versions of this method with different parameters.

Cursors

Queries always return a Cursor object. A Cursor is an object interface that provides random read-write access to the result set returned by a database query. It points to the first element in the result of the query.

A cursor is a pointer into a row of structured data. You can think of it as a pointer to table rows.

The Cursor class provides methods for moving the cursor through that structure, and methods to get the data from the columns of each row.

When a method returns a Cursor object, you iterate over the result, extract the data, do something with the data, and finally close the cursor to release the memory.

You will learn more about cursors in the following chapters.

Learn more

- SQLite website

- Full description of the query language
- SQLiteDatabase class
- Cursor class

10.1: SQLite Database

Contents:

- Using SQLite databases with Android
- Cursor
- ContentValues
- Implementing an SQLite database
- Database operations
- Instantiate Open Helper
- Working with the database
- Transactions
- Backing up databases
- Shipping a database with your APK
- Related practical
- Learn more

This chapter discusses the Android framework's SQLiteDatabase and SQLiteOpenHelper classes. It is not an introduction to SQLite or SQL databases.The chapter assumes that you are familiar with SQL databases in general, and basic SQL query building. Check out the SQL Primer chapter if you need a refresher.

Of the many storage options discussed, using a SQLite database is one of the most versatile, and straightforward to implement.

- An SQLite database is a good storage solution when you have structured data that you need to store persistently and access, search, and change frequently.
- You can use the database as the primary storage for user or app data, or you can use it to cache and make available data fetched from the cloud.
- If you can represent your data as rows and columns, consider a SQLite database.
- Content providers, which will be introduced in a later chapter, work excellently with SQLite databases.

When you use an SQLite database, represented as an SQLiteDatabase object, all interactions with the database are through an instance of the SQLiteOpenHelper class which executes your requests and manages your database for you. Your app should only interact with the SQLiteOpenHelper, which will be described below.

There are two data types associated with using SQLite databases in particular, Cursor and ContentValues.

Cursor

The SQLiteDatabase always presents the results as a Cursor in a table format that resembles that of a SQL database.

You can think of the data as an array of rows. A cursor is a pointer into one row of that structured data. The Cursor class provides methods for moving the cursor through the data structure, and methods to get the data from the fields in each row.

The Cursor class has a number of subclasses that implement cursors for specific types of data.

- SQLiteCursor exposes results from a query on a SQLiteDatabase. SQLiteCursor is not internally synchronized, so code using a SQLiteCursor from multiple threads should perform its own synchronization when using the SQLiteCursor.
- MatrixCursor is an all-rounder, a mutable cursor implementation backed by an array of objects that automatically expands internal capacity as needed.

Some common operations on cursor are:

- getCount() returns the number of rows in the cursor.
- getColumnNames() returns a string array holding the names of all of the columns in the result set in the order in which

they were listed in the result.
- getPosition() returns the current position of the cursor in the row set.
- Getters are available for specific data types, such as getString(int column) and getInt(int column).
- Operations such as moveToFirst() and moveToNext() move the cursor.
- close() releases all resources and makes the cursor completely invalid. Remember to call close to free resources!

Processing cursors

When a method call returns a cursor, you iterate over the result, extract the data, do something with the data, and finally, you must close the cursor to release the memory. Failing to do so can crash your app when it runs out of memory.

The cursor starts before the first result row, so on the first iteration you move the cursor to the first result if it exists. If the cursor is empty, or the last row has already been processed, then the loop exits. Don't forget to close the cursor once you're done with it. (This cannot be repeated too often.)

```
// Perform a query and store the result in a Cursor
Cursor cursor = db.rawQuery(...);
try {
    while (cursor.moveToNext()) {
        // Do something with the data
     }
} finally {
    cursor.close();
}
```

When you use a SQL database, you can implement your SQLiteOpenHelper class to return the cursor to the calling activity or adapter, or you can convert the data to a format that is more suitable for the adapter. The advantage of the latter is that managing the cursor (and closing it) is handled by the open helper, and your user interface is independent of what happens at the backend. See the SQLite Database practical for an implementation example.

ContentValues

Similar to how extras stores data, an instance of ContentValues stores data as key-value pairs, where the key is the name of the column and the value is the value for the cell. One instance of ContentValues represents one row of a table.

The insert() method for the database requires that the values to fill a row are passed as an instance of ContentValues.

```
ContentValues values = new ContentValues();
// Insert one row. Use a loop to insert multiple rows.
values.put(KEY_WORD, "Android");
values.put(KEY_DEFINITION, "Mobile operating system.");

db.insert(WORD_LIST_TABLE, null, values);
```

Implementing an SQLite database

To implement a database for your Android app, you need to do the following.

1. (Recommended) Create a data model.
2. Subclass SQLiteOpenHelper
    i. Use constants for table names and database creation query
    ii. Implement onCreate to create the SQLiteDatabase with tables for your data
    iii. Implement onUpgrade()
    iv. Implement optional methods

3. Implement the query(), insert(), delete(), update(), count() methods in SQLiteOpenHelper.
4. In your MainActivity, create an instance of SQLiteOpenHelper.
5. Call methods of SQLiteOpenHelper to work with your database.

Caveats:

- When you implement the methods, always put database operations into try/catch blocks.
- The sample apps do not validate the user data. When you write an app for publication, always make sure user data is what you expect to avoid the injection of bad data or execution of malicious SQL commands into your database.

Data model

It is a good practice to create a class that represents your data with getters and setters.

For an SQLite database, an instance of this class could represent one record, and for a simple database, one row in a table.

```
public class WordItem {
    private int mId;
    private String mWord;
    private String mDefinition;
    // Getters and setters and more
}
```

Subclass SQLiteOpenHelper

Any open helper you create must extend SQLiteOpenHelper.

```
public class WordListOpenHelper extends SQLiteOpenHelper {

    public WordListOpenHelper(Context context) {
       super(context, DATABASE_NAME, null, DATABASE_VERSION);
       Log.d(TAG, "Construct WordListOpenHelper");
    }
}
```

Define constants for table names

While not required, it is customary to declare your table, column, and row names as constants. This makes your code a lot more readable, makes it easier to change names, and your queries will end up looking a lot more like SQL. You can do this in the open helper class, or in a separate public class; you will learn more about this in the chapter about content providers.

```
    private static final int DATABASE_VERSION = 1;
  // has to be 1 first time or app will crash
  private static final String WORD_LIST_TABLE = "word_entries";
  private static final String DATABASE_NAME = "wordlist";

  // Column names...
  public static final String KEY_ID = "_id";
  public static final String KEY_WORD = "word";

  // ... and a string array of columns.
  private static final String[] COLUMNS = {KEY_ID, KEY_WORD};
```

Define query for creating database

You need a query that creates a table to create a database. This is also customarily defined as a string constant. This basic example creates one table with a column for an auto-incrementing id and a column to hold words.

```
    private static final String WORD_LIST_TABLE_CREATE =
            "CREATE TABLE " + WORD_LIST_TABLE + " (" +
             KEY_ID + " INTEGER PRIMARY KEY, " +
             // will auto-increment if no value passed
             KEY_WORD + " TEXT );";
```

Implement onCreate() and create the database

The onCreate method is only called if there is no database. Create your tables in the method, and optionally add initial data.

```
@Override
public void onCreate(SQLiteDatabase db) { // Creates new database
   db.execSQL(WORD_LIST_TABLE_CREATE); // Create the tables
   fillDatabaseWithData(db); // Add initial data
   // Cannot initialize mWritableDB and mReadableDB here, because
   // this creates an infinite loop of on Create()
   // being repeatedly called.
}
```

Implement onUpgrade()

This is a required method.

If your database acts only as a cache for data that is also stored online, you can drop the the tables and recreate them after the upgrade is complete.

Note: If your database is the main storage, you **must** preserve the user's data before you do this as this operation destroys all the data. See the chapter on Storing Data.

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // SAVE USER DATA FIRST!!!
        Log.w(WordListOpenHelper.class.getName(),
                "Upgrading database from version " + oldVersion + " to "
                        + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + WORD_LIST_TABLE);
        onCreate(db);
}
```

Optional methods

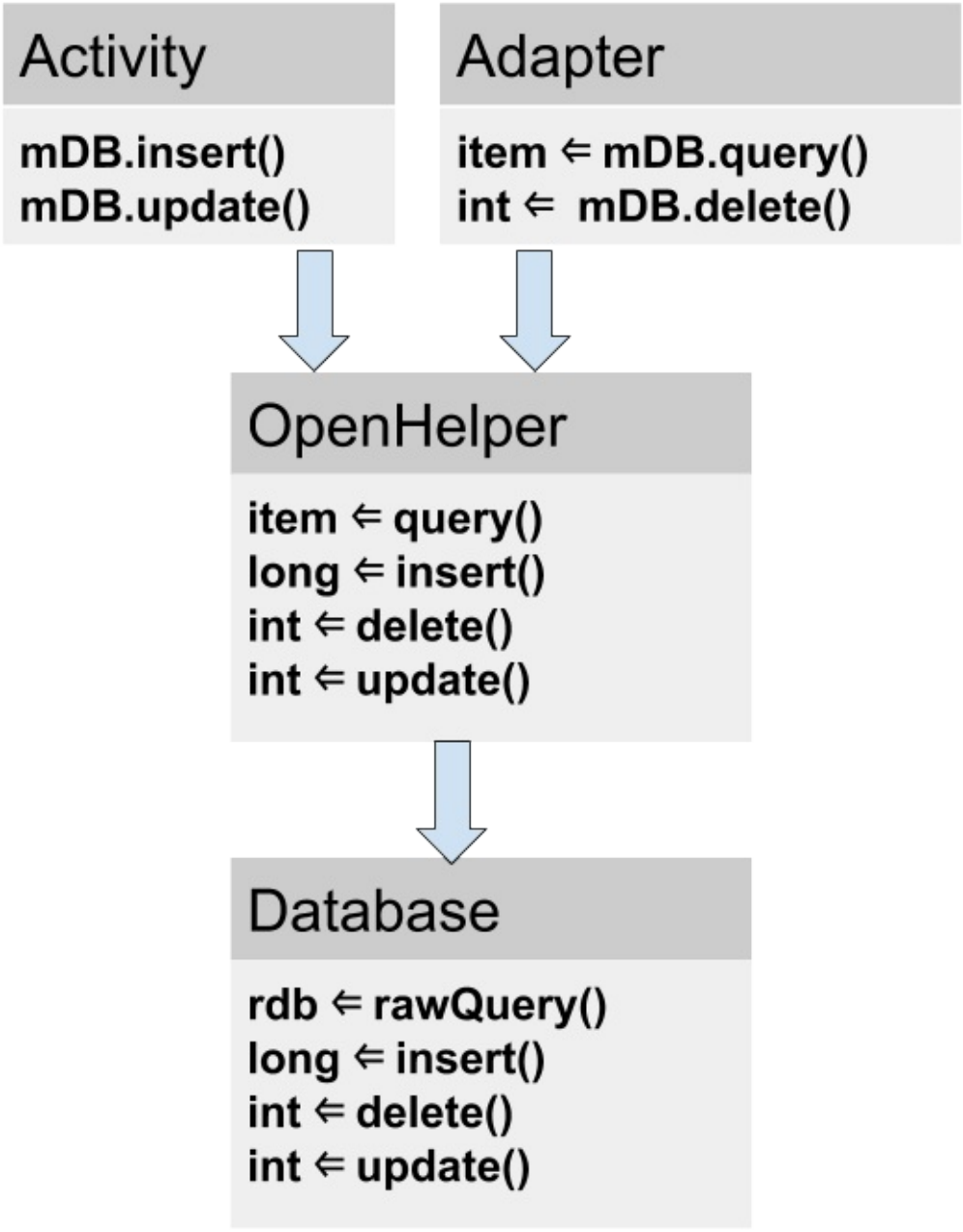
The open helper class provides additional methods that you can override as needed.

- onDowngrade()—The default implementation rejects downgrades.
- onConfigure()—called before onCreate. Use this only to call methods that configure the parameters of the database connection.
- onOpen()—Any work other than configuration that needs to be done after the database is opened.

Database operations

While you can call your methods in the open helper anything you want and have them return anything you choose to the calling activity, it is a good idea to go with the standardized query(), insert(), delete(), update(), count() methods that match the API of the database and content providers. Using this format will make it easier to add a content provider or loader in the future, and it makes it easier for other people to understand your code.

The following diagram shows how the different API's should be designed for consistency and clarity.



query()

The query method that you implement in your open helper class can take and return any data type that your user interface needs.

Since the open helper provides convenience methods for inserting, deleting, and updating rows, your query method does not need to be generic and support these operations.

In general, your query method should only allow queries that are needed by your app and not be general purpose.

The database provides two methods for sending queries: SQLiteDatabase.rawQuery() and SQLiteDatabase.query(), with several options for the arguments.

SQLiteDatabase.rawQuery()

The open helper query method can construct an SQL query and send it as a rawQuery to the database which returns a cursor. If your data is supplied by your app, and under your full control, you can use rawQuery().

```
rawQuery(String sql, String[] selectionArgs)
```

- The first parameter to db.rawquery() is an SQLite query string.
- The second parameter contains the arguments.

```
cursor = mReadableDB.rawQuery(queryString, selectionArgs);
```

SQLiteDatabase.query()

If you are processing user-supplied data, even after validation, it is more secure to construct a query and use a version of the SQLiteDatabase.query() method for the database. The arguments are what you'd expect in SQL and are documented in the SQLiteDatabase documentation.

```
Cursor query (boolean distinct,  String table, String[] columns, String selection,
                String[] selectionArgs, String groupBy, String having,
                String orderBy,String limit)
```

Here is a basic example:

```
String[] columns = new String[]{KEY_WORD};
String where =  KEY_WORD + " LIKE ?";
searchString = "%" + searchString + "%";
String[] whereArgs = new String[]{searchString};
cursor = mReadableDB.query(WORD_LIST_TABLE, columns, where, whereArgs, null, null, null);
```

Example of complete open helper query()

```
public WordItem query(int position) {
   String query = "SELECT  * FROM " + WORD_LIST_TABLE +
           " ORDER BY " + KEY_WORD + " ASC " +
           "LIMIT " + position + ",1";

   Cursor cursor = null;
   WordItem entry = new WordItem();

   try {
       if (mReadableDB == null) {mReadableDB = getReadableDatabase();}
       cursor = mReadableDB.rawQuery(query, null);
       cursor.moveToFirst();
       entry.setId(cursor.getInt(cursor.getColumnIndex(KEY_ID)));
       entry.setWord(cursor.getString(cursor.getColumnIndex(KEY_WORD)));
   } catch (Exception e) {
       Log.d(TAG, "EXCEPTION! " + e);
   } finally {
       // Must close cursor and db now that we are done with it.
       cursor.close();
       return entry;
   }
}
```

insert()

The open helper's insert() method calls SQLiteDatabase.insert(), which is a SQLiteDatabase convenience method to insert a row into the database. (It's a convenience method, because you do not have to write the SQL query yourself.)

Format

```
long insert(String table, String nullColumnHack, ContentValues values)
```

- The first argument is the table name.
- The second argument is a `String nullColumnHack`. It's a workaround that allows you to insert empty rows. See the documentation for insert(). Use null.
- The third argument must be a ContentValues container with values to fill the row. This sample only has one column; for tables with multiple columns, you add the values for each column to this container.
- The database method returns the id of the newly inserted item, and you should pass that on to the application.

Example

```
newId = mWritableDB.insert(WORD_LIST_TABLE, null, values);
```

delete()

The open helper delete method calls the databases delete() method, which is a convenience method so that you do not have to write the full SQL query.

Format

```
int delete (String table, String whereClause, String[] whereArgs)
```

- The first argument is the table name.
- The second argument is a WHERE clause.
- The third argument are the arguments to the WHERE clause.

You can delete using any criteria, and the method returns the number of items that were actually deleted, which the open helper should return also.

Example

```
deleted = mWritableDB.delete(WORD_LIST_TABLE,
                    KEY_ID + " =? ", new String[]{String.valueOf(id)});
```

update()

The open helper update method calls the database's update() method, which is a convenience method so that you do not have to write the full SQL query. The arguments are familiar from previous methods, and the onUpdate returns the number of rows updated.

Format

```
int update(String table, ContentValues values,
    String whereClause, String[] whereArgs)
```

- The first argument is the table name.
- The second argument must be a ContentValues with new values for the row.
- The third argument is a WHERE clause.
- The fourth argument are the arguments to the WHERE clause.

Example

```
ContentValues values = new ContentValues();
values.put(KEY_WORD, word);
mNumberOfRowsUpdated = mWritableDB.update(WORD_LIST_TABLE,
values, // new values to insert
KEY_ID + " = ?",
new String[]{String.valueOf(id)});
```

count()

The count() method returns the number of entries in the database. If you are using a RecyclerView.Adapter, it has to implement getItemCount(), which needs to get the number of rows from the open helper which needs to get it from the database.

In adapter

```
@Override
public int getItemCount() {
    return (int) mDB.count();
}
```

In the open helper

```
public long count(){
    if (mReadableDB == null) {mReadableDB = getReadableDatabase();}
    return DatabaseUtils.queryNumEntries(mReadableDB, WORD_LIST_TABLE);
}
```

queryNumEntries()) is a method in the public DatabaseUtils class, which provides many convenience methods for working with cursors, databases, and also content providers.

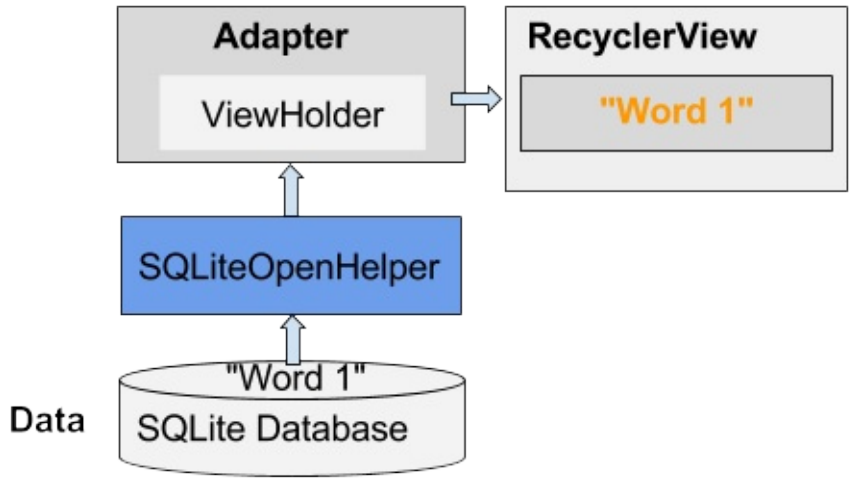
Instantiate Open Helper

To get a handle to the database, In MainActivity, in onCreate, call:

```
mDB = new WordListOpenHelper(this);
```

Working with the database

It is a common pattern to combine a SQLiteDatabase backend with a RecyclerView to display the data.



For example:

- Pressing the FAB could start an activity that gets input from the user and stores it into the database as a new or updated item.
- Swiping an item might delete it after the user confirms deletion.

Transactions

Use transactions

- when performing multiple operations that all need to complete to keep database consistent, for example, updating pricing of related items for a sale event.
- to batch multiple independent operations to improve performance, such as mass inserts.

Transactions can be nested, and the SQLiteDatabase class provides additional methods to manage nested transactions. See SQLiteDatabase references documentation.

Transaction idiom

```
db.beginTransaction();
try {
  ...
  db.setTransactionSuccessful();
} finally {
  db.endTransaction();
}
```

Backing up databases

It is a good idea to back up your app's database.

You can do so using the Cloud Backup options discussed in the Storage Options chapter.

Shipping a database with your app

Sometimes you may want to include a populated database with your app. There are several ways in which to do that, and there are trade-offs for each.

- Include the SQL commands with the application and have it create the database and insert the data on first use. This is basically what you will do in the practical for data storage. If the amount of data you want put in the database is small, just an example so that the user gets to see something, you can use this method.
- Ship the data with the APK as a resource, and build the database when the user opens the app for the first time. This is similar to the first method, but instead of defining your data in your code, you put it in a resource, for example, in CSV format. You can then read the data with an input stream and add it to the database.
- Build and pre-populate the SQLite database and include it in the APK. With this method you write an app that creates and populates a database. You can do this on the emulator. You then copy the file in which your database is actually stored ("/data/data/YOUR_PACKAGE/databases/" directory) and include it as an asset with your app. When the app is started for the first time, you copy the database file back into the "/data/data/YOUR_PACKAGE/databases/" directory.

The SQLiteAssetHelper class, which you can download from Github, extends SQLiteOpenHelper to help you do this. And this Stackoverflow post discusses this topic in more detail.

Note that for a larger database, populating the database should be done in the background, and your app should not crash if there is no database yet, or the database is empty.

Related practical

The related practical documentation is in Android Developer Fundamentals: Practicals.

- SQLite Data Storage
- Searching a SQLite Database

Learn more

- Storage Options
- Saving Data in SQL Databases
- SQLiteDatabase class
- ContentValues class
- SQLiteOpenHelper class
- Cursor class
- SQLiteAssetHelper class from Github

11.1: Share Data Through Content Providers

Contents:

- What is a Content Provider
- What is a Content Resolver
- Example of an app sharing data using a Content Provider
- What Content Providers are good for
- App Architecture with a Content Provider
- Implementing a Content Provider
- Data
- Contract
- Methods: insert, delete, update, query
- query() method
- Using a Content Resolver
- Permissions for sharing
- Related practical
- Learn more

What is a Content Provider?

A ContentProvider is a component that interacts with a repository. The app doesn't need to know where or how the data is stored, formatted, or accessed.

A content provider:

- Separates data from the app interface code
- Provides a standard way of accessing the data
- Makes it possible for apps to share data with other apps
- Is agnostic to the repository, which could by a database, a file system, or the cloud.

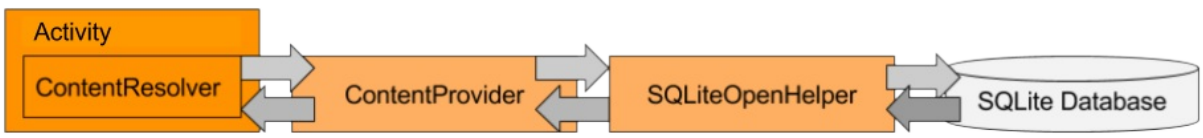
What is a Content Resolver?

To get data and interact with a content provider, an app uses a ContentResolver to send requests to the content provider.

The ContentResolver object provides query(), insert(), update(), and delete() methods for accessing data from a content provider.

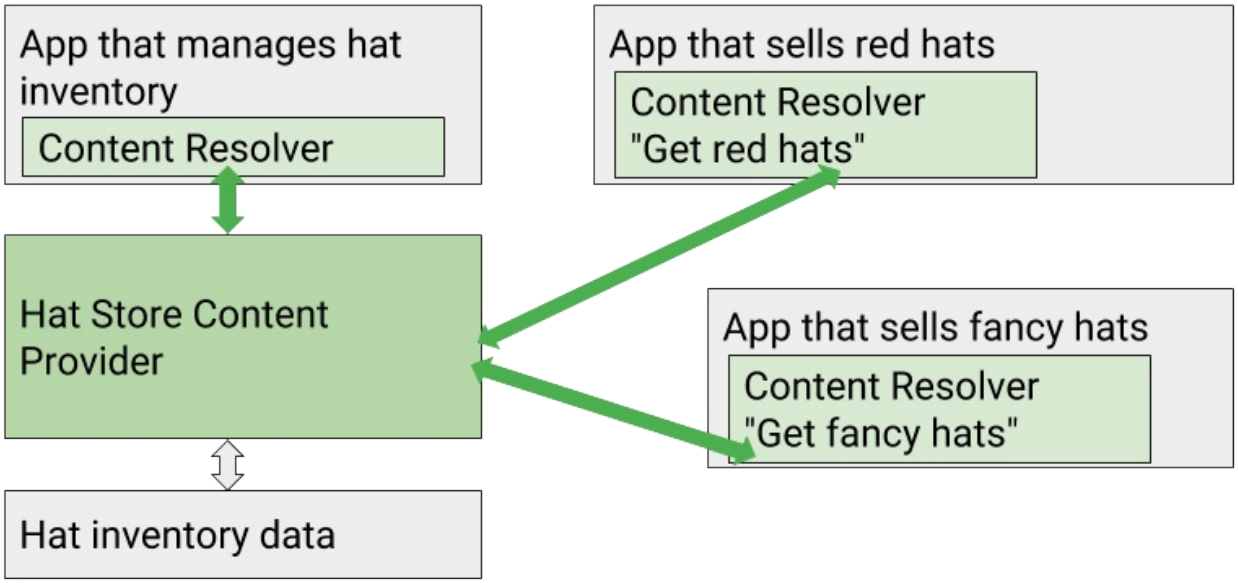
Each request consists of a URI and a SQL-like query, and the response is a Cursor object.

Note: You learned about cursors in the Data Storage chapters, and there is a recap later in this chapter.
The following diagram shows the query flow from an activity using a content resolver to the content provider to data in a SQL database, and back. Note that storing the data in a SQLite database is common, but not required.



Example of an app sharing data using a Content Provider

Consider an app that keeps an inventory of hats and makes it available to other apps that want to sell hats. The app that owns the data manages the inventory, but does not have a customer-facing interface. Two apps, one that sells red hats and one that sells fancy hats, access the inventory repository, and each fetch data relevant for their shopping apps.



What Content Providers are good for

Content providers are useful for apps that want to make data available to other apps.

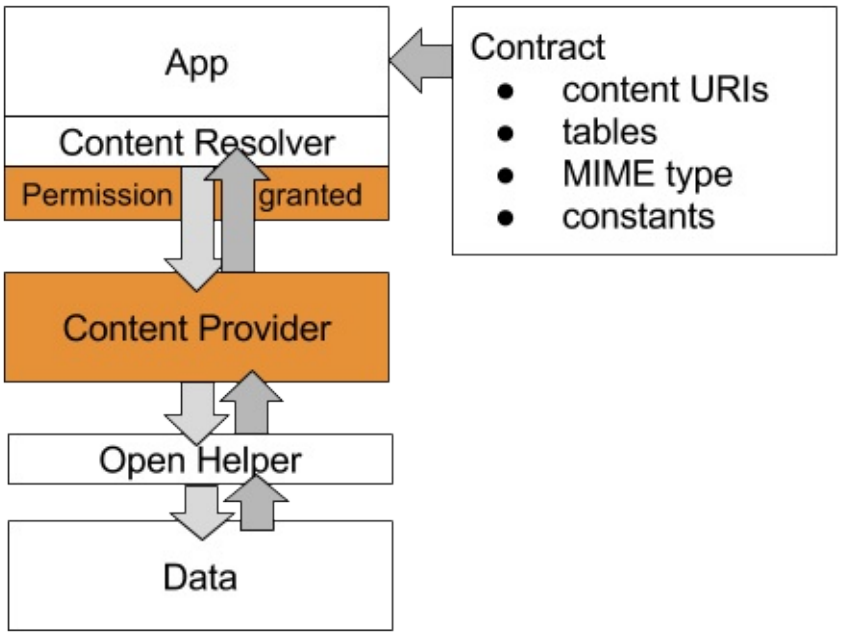
- With a content provider, you can allow multiple other apps to securely access, use, and modify a single data source that your app provides. Examples: Warehouse inventory for retail stores, game scores, or a collection of physics problems for colleges.
- For access control, you can specify levels of permissions for your content provider, specifying how other apps can access the data. For example, stores may not be allowed to change the warehouse inventory data.
- You can store data independently from the app, because the content provider sits between your user interface and your stored data. You can change how the data is stored without needing to change the user-facing code. For example, you can build a prototype of your shopping app using mock inventory data, then later replace it with an SQL database for the real data. You could even store some of your data in the cloud and some locally, and it would be all the same to your users.
- Another benefit of separating data from the user interface with a content provider is that development teams can work independently on the user interface and data repository of your app. For larger, complex apps it is very common that the user interface and the data backend are developed by different teams, and they can even be separate apps; that is, it is not required that the app with the content provider have a user interface. For example, your inventory app could consist only of the data and the content provider.
- There are other classes that expect to interact with a content provider. For example, you must have a content provider to use a loader, such as CursorLoader, to load data in the background. You will learn about loaders in the next chapter.

Note: If your app is the only one using the data, and you are developing all of it by yourself, you probably don't need a content provider.

App Architecture with a Content Provider

Architecturally, the content provider is a layer between the content-providing app's data storage backend and the rest of the app, separating the data and the interface.

To give you a picture of the whole content provider architecture, this section shows and summarizes all the parts of the implemented content provider architecture, as shown in the following diagram. Each part will be discussed in detail next.



Data and Open Helper: The data repository. The data could be in a database, a file, on the internet, generated dynamically, or even a mix of these. For example, if you had a dictionary app, the base dictionary could be stored in a SQLite database on the user's device. If a definition is not in the database, it could get fetched from the internet, and if that fails, too, the app could ask the user to provide a definition or check their spelling.

Data used with content providers is commonly stored in SQLite databases, and the content provider API mirrors this assumption.

Contract: The contract is a public class that exposes important information about the content provider to other apps. This usually includes the URI schemes, important constants, and the structure of the data that will be returned. For example, for the hat inventory app, the contract could expose the names of the columns that contain the price and name of a product, and the URI for retrieving an inventory item by part number.

Content Provider: The content provider extends the ContentProvider class and provides query(), insert(), update(), and delete() methods for accessing the data. In addition, it provides a public and secure interface to the data, so that other apps can access the data with the appropriate permissions. For example, to get inventory information from your app's database, the retail hat app would connect to the content provider, not to the database directly, as that is not permitted.

The app that owns the data specifies what permissions other apps need to have to work with the content provider. For example, if you have an app that provides inventory to retail stores, your app owns the data and determines the access permissions of other apps to the data. Permissions are specified in the Android Manifest.

Content Resolver: Content providers are always accessed through a content resolver. Think of the content resolver as a helper class that manages all the details of connecting to a content provider for you. Mirroring the content provider's API, the ContentResolver object provides you with query(), insert(), update(), and delete() methods for accessing data of a content provider. For example, to get get all the inventory items that are red hats, a hat store app would build a query for red hats, and use a content resolver to send that query to the content provider.

Implementing a Content Provider

Referring to the previous diagram, to implement a content provider you need:

- Data, for example, in a database.
- A way for accessing the data storage, for example, through an open helper for a database.
- A declaration of your content provider in the Android Manifest to make it available to your own and other apps.

- The subclass of ContentProvider that implements the query(), insert(), delete(), update(), count(), and getType() methods.
- Public contract class that exposes the URI scheme, table names, MIME type, and important constants to other classes and apps. While this is not mandatory, without it, other apps cannot know how to access your content provider.
- A content resolver to access the content provider using the appropriate methods and queries.

Let's take a look at each of these components.

Data

The data is often stored in a SQLite database, but this is not mandatory. Data could be stored in a file or file system, on the internet, or created dynamically. Or even a mix of these options. To the app, the content resolver the fetched data always as a Cursor object, as if it all came from the same source and in the same format.

The content provider might access the data directly in the case of files, or it might do so through a helper class. For example, apps typically use an open helper to interact with a SQLite database, and the content provider interacts with the open helper to get the data.

Typically, the data is presented to the content provider by the data store as tables, similar to database tables, where each row represents one entry, and each column represents an attribute for that entry. For example, each row contains one contact, and may have columns for email addresses and phone numbers. The structure of the tables is exposed in the contract.

Note: If you are just working with files, you can use the predefined FileProvider class.

Contract

The contract is a public class that exposes important information about an app's content provider so that other apps know how to access and use the content provider.

Using a contract separates public from private app information, design from implementation, and gives other apps one place to get all the information they need to work with a content provider. While the underlying app may change, a contract defines an API that ideally does not change once the app is published.

The contract for a content provider typically includes:

- **Content URI and URI scheme.** The URI scheme shows how to build URIs to access the content provider's data. It's the API for the data.
- **Table constants.** Makes table and column names available as constants, because they are needed to extract data from the returned cursor object.
- **MIME types**, which have information on the data format, so that the app can appropriately process returned data. For example, that data could be encoded in JSON or HTML, or use a custom format.
- Other shared constants that make it more convenient for an app to use the content provider.

Note: Contracts are not limited to content providers. You can use a contract anytime you want to share constants across classes of your app or make information about your app available to other apps.

URI Scheme & Content URI

Apps send requests to the content provider using Uniform Resource Identifiers or URIs.

A content URI for content providers has this general form:

```
scheme://authority/path/ID
```

- **scheme** is always **content://** for content URIs.

- **authority** represents the domain, and for content providers customarily ends in `.provider`
- **path** is the path to the data.
- **ID** uniquely identifies the data set to search.

For example, the following URI could be used to request all the entries in the "words" table:

```
content://com.android.example.wordcontentprovider.provider/words
```

The URI scheme for a content provider is defined in the Contract so that it is available to any app that wants to query this content provider. Customarily, this is done defining constants for AUTHORITY, CONTENT_PATH, and CONTENT_URI.

- AUTHORITY. Represents the domain. For content providers this includes the unique package name and ends in `.provider`
- CONTENT_PATH. The content path is an abstract semantic identifier of the data you are interested in. It does not predict or presume in what form the data is stored or organized in the background. As such, the path could resolve in the name of a table, the name of a file, or the name of the list.
- CONTENT_URI. This is a content:// style URI to one set of data. If you have multiple "data containers" in the backend, you would create a content URI for each. For example, you would create a content URI for each table that can be queried. Use the Uri helper class for building and manipulating URIs.

In code, this might look as follows:

```
public static final String AUTHORITY =
"com.android.example.minimalistcontentprovider.provider";

public static final String CONTENT_PATH =  "words";
public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY +
"/" + CONTENT_PATH);
```

Tables in the Contract

A common way of organizing a contract class is to put definitions that are global to your database into the root level of the class. Usually, this is the name of the database.

Create a static abstract inner class for each table with the column names. This inner class commonly implements the BaseColumns interface. By implementing the BaseColumns interface, your class can inherit a primary key field called _ID that some Android classes, such as cursor adapters, expect to exist. This is not required, but can help your database work harmoniously with the Android framework.

Your code in the contract might look like this:

```
public static final String DATABASE_NAME = "wordlist";

public static abstract class WordList implements BaseColumns {
    public static final String WORD_LIST_TABLE = "word_entries";
    // Column names...
    public static final String KEY_ID = "_id";
    public static final String KEY_WORD = "word"
}
```

MIME Type

The MIME type tells an app, what type and format received data is in, so that it can process the data appropriately. Common MIME types include `text/html` for web pages, and `application/json`. If your content provider returns data in either of those standard formats, you should use the standard MIME types. A full list of these standard types is available on the IANA MIME Media Types website.

However, your content provider probably returns data specific to your app. In that case, you will need to specify a custom MIME type.

For content URIs that point to a row or rows of table data, and that are thus unique to your app, the MIME type should be in Android's vendor-specific MIME format. The general format is:

```
type.subtype/provider-specific-part
```

Where the parts should be:

- Type part: vnd
- Subtype part:
  - If the URI pattern is for a single row: android.cursor.item/
  - If the URI pattern is for more than one row: android.cursor.dir/
- Provider-specific part: vnd..
- You supply the and .
  - The value should be globally unique. A good choice for is your company's name or some part of your application's Android package name.
  - The value should be unique to the corresponding URI pattern. A good choice for the is a string that identifies the table associated with the URI.

For example, if a provider's authority is com.example.app.provider, and it exposes a table named "words", the MIME type for multiple rows in "words" is:

```
vnd.android.cursor.dir/vnd.com.example.provider.words
```

For a single row of "words", the MIME type is:

```
vnd.android.cursor.item/vnd.com.example.provider.words
```

And you would specify it in your contract as:

```
static final String SINGLE_RECORD_MIME_TYPE =
        "vnd.android.cursor.item/vnd.com.example.provider.words";
static final String MULTIPLE_RECORDS_MIME_TYPE =
        "vnd.android.cursor.item/vnd.com.example.provider.words";
```

An app can call the getType() method of a content provider to find out what type of data to expect. Your getType() method might look like this:

```
@Override
public String getType(Uri uri) {
   switch (sUriMatcher.match(uri)) {
       case URI_ALL_ITEMS_CODE:
           return MULTIPLE_RECORDS_MIME_TYPE;
       case URI_ONE_ITEM_CODE:
           return SINGLE_RECORD_MIME_TYPE;
       default:
           return null;
   }
}
```

Read more about MIME types for content providers in the Android developer documentation.

Note: The MIME type does not tell the clients how to process that data. As such, the custom MIME type only provides a hint, and the contract should provide additional information to the client as to what the expected data formats are.

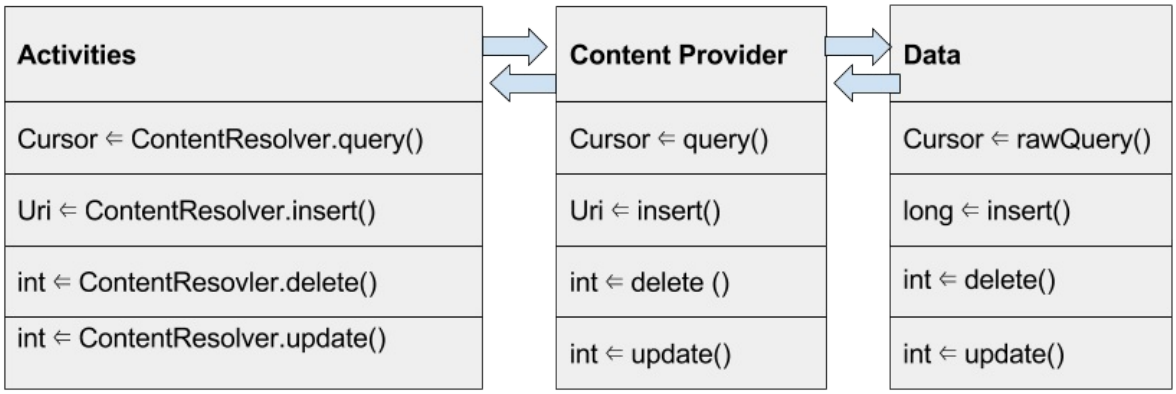
Methods: insert, delete, update, query

The content resolver makes available query, insert, delete, and update methods, and the content provider implements those same methods to access the data.

As such, it is a good practice to keep the names, method arguments, and return values consistent between all components. This makes implementation and maintenance a lot easier.

The following diagram shows the APIs between the conceptual building blocks of an app that uses a content provider to access data. Note in particular:

- The methods are named the same and return the same data type throughout the stack (except for insert()).
- Because it is common for a content provider to connect to a database, the query method returns a cursor. If your backend is not a database, the content provider must do the work of converting the returned data into the cursor format.
- This diagram does not show additional helper classes, such as open helpers for databases, that may also use the same API convention.



When you create a content provider by extending the ContentProvider class, you need to implement the insert, delete, update, and query methods. If you follow the principle of making the method signatures the same across components, passing data back and forth does not require a lot of code.

Here is are sample methods in a content provider. Note that the content provider receives the values to insert in the correct type, as ContentValues, calls the database, and builds and returns the required Uri for the content resolver.

Insert method

```
/**
* Inserts one row.
*
* @param uri Uri for insertion.
* @param values Container for Column/Row key/value pairs.
* @return URI for the newly created entry.
*/
@Override
public Uri insert(Uri uri, ContentValues values) {
   long id = mDB.insert(values);
   return Uri.parse(CONTENT_URI + "/" + id);
}
```

Delete method

```
/**
* Deletes records(s) specified by selectionArgs.
*
* @param uri URI for deletion.
* @param selection Where clause.
* @param selectionArgs Where clause arguments.
* @return Number of records affected.
*/
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
   return mDB.delete(parseInt(selectionArgs[0]));
}
```

Update method

```
/**
* Updates records(s) specified by selection/selectionArgs combo.
*
* @param uri URI for update.
* @param values Container for Column/Row key/value pairs.
* @param selection Where clause.
* @param selectionArgs Where clause arguments.
* @return Number of records affected.
*/
@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
   return mDB.update(parseInt(selectionArgs[0]), values.getAsString("word"));
}
```

query() method

The query method in the content provider has the following signature:

```
public Cursor query(Uri uri, String[] projection, String selection,String[] selectionArgs, String sortOrder){}
```

The arguments represent the parts of an SQL query and are discussed below. The query method of the content provider must parse the URI argument and determine the appropriate action.

URI Matching

It is a good practice to use an instance of the UriMatcher class to match the URIs. UriMatcher is a helper class for matching URIs for content providers.

1. Create a new UriMatcher in your content provider.

   ```
   private static UriMatcher sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
   ```

2. In onCreate() add the URIs to match the the matcher. These are the content URIs defined in the Contract. You may want to do this in a separate method, initializeUriMatching() that you call in onCreate(). The example code includes URIs requesting all items, one item by ID, and the item count.

```
/**
* Defines the accepted Uri schemes for this content provider.
* Calls addURI() for all of the content URI patterns that the provider should recognize.
*/
private void initializeUriMatching() {

   // Matches a URI that is just the authority + the path,
   // triggering the return of all words.
   sUriMatcher.addURI(AUTHORITY, CONTENT_PATH, URI_ALL_ITEMS_CODE);

   // Matches a URI that references one word in the list by its index.
   sUriMatcher.addURI(AUTHORITY, CONTENT_PATH + "/#", URI_ONE_ITEM_CODE);

   // Matches a URI that returns the number of rows in the table.
   sUriMatcher.addURI(AUTHORITY, CONTENT_PATH + "/" + COUNT, URI_COUNT_CODE);
}
```

The query method switches on the matching URI to query the database for all items, one item, or an item count, as shown in this example code.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
                    String sortOrder) {

   Cursor cursor = null;

   // Determine integer code from the URI matcher and switch on it.
   switch (sUriMatcher.match(uri)) {
       case URI_ALL_ITEMS_CODE:
           cursor =  mDB.query(ALL_ITEMS);
           Log.d(TAG, "case all items " + cursor);
           break;
       case URI_ONE_ITEM_CODE:
           cursor =  mDB.query(parseInt(uri.getLastPathSegment()));
           Log.d(TAG, "case one item " + cursor);
           break;
       case URI_COUNT_CODE:
           cursor = mDB.count();
           Log.d(TAG, "case count " + cursor);
           break;
       case UriMatcher.NO_MATCH:
           // You should do some error handling here.
           Log.d(TAG, "NO MATCH FOR THIS URI IN SCHEME: " + uri);
           break;
       default:
           // You should do some error handling here.
           Log.d(TAG, "INVALID URI - URI NOT RECOGNIZED: "  + uri);
   }
   return cursor;
}
```

Using a Content Resolver

The ContentResolver object provides methods to query(), insert(), delete(), and update() data. Thus, the content resolver mirrors the content providers API and manages all interaction with the content provider for you. In most situations, you can use the default content resolver provided by the Android system.

Cursors

The content provider always presents the query results as a Cursor in a table format that resembles of a SQL database. This is independent of how the data is actually stored.

A cursor is a pointer into a row of structured data. You can think of it as a linked list of rows. The Cursor class provides methods for moving the cursor through that structure, and methods to get the data from the columns of each row.

When a method returns a cursor, you iterate over the result, extract the data, do something with the data, and finally close the cursor to release the memory.

If you use a SQL database, as shown above, you can implement your open helper to return a cursor, and then again, the content provider returns a cursor via the content resolver. If your data storage returns data in a different format, you will have to convert it into a cursor, usually a MatrixCursor.

The query() method

To make a query to the content provider:

1. Create an SQL-style query.
2. Use a content resolver to interact with the content provider to execute the query and return a Cursor.
3. Process the results in the Cursor.

The query method has the following signature:

```
public Cursor query(Uri uri, String[] projection, String selection,String[] selectionArgs, String sortOrder){}
```

The arguments to this method represent the parts of a SQL query. Even if you are using another kind of backend, you must still accept a query in this style and handle the arguments appropriately.

uri	The complete content URI queried. This cannot be null. You get the information for the correct URI from the contract. For example: `String queryUri = Contract.CONTENT_URI.toString();`
projection	A string array with the names of the columns to return for each row. Setting this to null returns all columns. For example: `String[] projection = new String[] {Contract.CONTENT_PATH};`
selection	Indicates which rows/records of the objects you want to access. This is a WHERE clause excluding the actual where. For example: `String where = KEY_WORD + " LIKE ?";`
selectionArgs	Argument values for the selection criteria. If you include ?s in selection, they are replaced by values from selectionArgs, in the order that they appear. IMPORTANT: It is a best security practice to always separate selection and selectionArgs. For example: `String[]whereArgs = new String[]{searchString};`
sortOrder	The order in which to sort the results. Formatted as an SQL ORDER BY clause (excluding the ORDER BY keyword). Usually ASC or DESC; null requests the default sort order, which could be unordered.

And you make a query to to the content provider like this:

```
Cursor cursor = getContentResolver().query(Uri.parse(queryUri), projection, selectionClause, selectionArgs, sortOrder
);
```

Note: The insert, delete, and update methods are provided for convenience and clarity. Technically, the query method could handle all requests, including those to insert, delete, and update data.

Permissions for sharing

By default, apps cannot access the data of other apps. Both apps involved in sharing data need to have permission to do so.

- The content provider must allow other apps to access it's data.
- The user must allow the client app to access the content provider.

Permissions in from the content provider

To make your content provider visible and available to other apps, you need to declare in the AndroidManifest of the provider.

```
<provider android:name=".WordListContentProvider" android:authorities="com.android.example.wordlistsqlwithcontentprov
ider.provider" android:exported="true"/>
```

The `android:exported` property makes it explicit that other apps can use this content provider.

With no permissions set explicitly, any other app can access the content provider for reading and writing. To limit and make explicit access constraints, set permissions inside the provider tag of the content provider, where myapp is the unique name of your app:

```
android:readPermission="com.android.example.wordlistsqlwithcontentpfonrovider.PERMISSION"
android:writePermission="com.android.example.wordlistsqlwithcontentprovider.PERMISSION"
```

- The permission string should be unique to your content provider, so that it only grants privileges for your content provider.
- While the string can be anything, using the package name guarantees uniqueness.

Permissions client app requests from user

In order to access the content provider, the client app needs to declare permissions in the Android Manifest for that content provider.

```
<uses-permission android:name = "com.android.example.wordlistsqlwithcontentprovider.PERMISSION"/>
```

Permissions are not covered in detail in these concepts.

You can learn more in Declaring Permissions, System Permissions, and Implementing Content Provider Permissions.

Related practical

The related practicals documentation is in Android Developer Fundamentals: Practicals.

- Minimalist Content Providers
- Add a Content Provider to WordListSQL
- Sharing Content with Other Apps

Learn more

Developer Documentation:

- Content Provider Basics
- Content Providers
- Uniform Resource Identifiers or URIs
- MIME type
- MatrixCursor and Cursors
- Working with System Permissions
- Implementing Content Provider Permissions

Videos:

- Android Application Architecture
- Android Application Architecture: The Next Billion Users

12.1: Loaders

Contents:

- Loader architecture
- Implementing a Cursor Loader
- Related practical
- Learn more

One of the major reasons users abandon apps is startup time. Research shows that if an app or page takes more than 3 seconds to load, 40% of users will abandon it. This number varies somewhat depending on the study, but the overwhelming fact is that user retention is strongly tied to app loading speed.

App load time is directly related to whatever happens on the UI thread. The less work your UI thread has to do, the faster your users will see the page. There are many factors that affect app startup time, and you will learn more about app performance in a later chapter. One of the big, obvious actions that affect performance is how long it takes for your app to load its data.

If you know exactly where your data is coming from, you can potentially optimize by loading it yourself. If your data is supplied by a content provider, you may not know what the backend is, and you may not know whether for any given user, there will be a small or large amount of data.

The solution is to load most or all of your data in the background, while you show your users relevant information that you have stored locally. For example, you could show them the latest cached weather information, until you have retrieved new data that shows the current weather for the current location.

Loaders are special purpose classes that manage loading and reloading updated data asynchronously in the background using AsyncTask.

Introduced in Android 3.0, loaders have these characteristics:

- They are available to every Activity and Fragment.
- They provide asynchronous loading of data in the background.
- They monitor the source of their data and automatically deliver new results when the content changes. For example, if you are displaying data in RecyclerView, when the underlying data changes, the a CursorLoader automatically loads an updated set of data, and when finished with loading, can notify the RecyclerView.Adapter to update what it displays to the user.
- Loaders automatically reconnect to the last loader's cursor when being recreated after a configuration change. Thus, they don't need to re-query their data to display it to you.

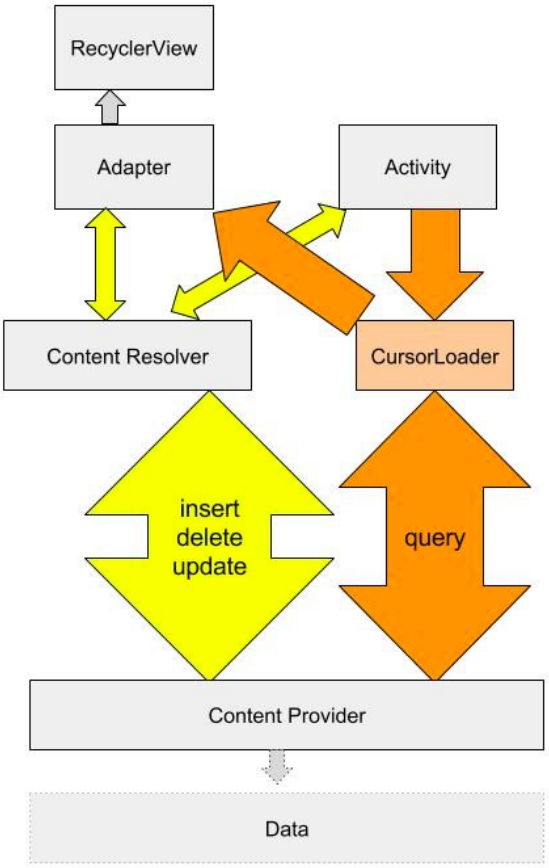
In a previous chapter you learned about AsyncTask as a general purpose class for doing work in the background, and you used an AsyncTaskLoader to keep data available to your users through configuration changes.

While you can create custom loaders by subclassing the Loader class, the Android framework provides CursorLoader that is straightforward to use and applies to many use cases. The CursorLoader extends AsyncTaskLoader to specifically work with content providers, saving you a lot of work.

Note that it is entirely possible to build custom loaders. But since the Android system provides you with an elegant solution that saves you a lot of work, consider how you can use it as given before implementing your own solution from scratch. Before writing your own loader, always consider whether you can improve your app design to work with a CursorLoader.

Loader architecture

As shown in the diagram below, the loader replaces the content resolvers query call to the content provider. The diagram shows a simplified version of app architecture with a loader. The loader performs querying for items in the background. It observes the data for you, and if the data changes, it automatically gets a new set of data and hands it to the adapter.



Implementing a CursorLoader

An application that uses loaders typically includes the following:

- An Activity or Fragment.
- An instance of the LoaderManager.
- A CursorLoader to load data backed by a ContentProvider. Alternatively, you can implement your own subclass of Loader or AsyncTaskLoader to load data from some other source.
- An implementation for LoaderManager.LoaderCallbacks. This is where you create new loaders and manage your references to existing loaders.
- A way of displaying the loader's data, such as a SimpleCursorAdapter or RecyclerViewAdapter.
- A data source, such as a ContentProvider (with a CursorLoader).

LoaderManager

The LoaderManager is a convenience class that manages all your loaders. You only need one loader manager per activity and typically get it in onCreate() of your activity, where you also register the loaders you are going to use.

The loader manager takes care of registering an observer with the content provider, which receives callbacks when data in the content provider changes.

The only calls to the loader manager you need to make are for registering a loader, and restarting it when you need to discard all the loaded data. The first parameter is the ID of the loader, the second is optional arguments, and the third is the context where the callbacks are defined.

```
getLoaderManager().initLoader(0, null, this);
getLoaderManager().restartLoader(0, null, this);
```

LoaderManager.LoaderCallbacks

In order to interact with the loader, you activity has to implement a set of callbacks specified in the LoaderCallbacks interface of the LoaderManager. When the state of the loader changes, these methods are called accordingly. The methods are:

- onCreateLoader()—Called when a new loader is created. Associates loader with the data source it should load and observe. (You don't have to do anything additional for the loader to observer your data source.)
- onLoadFinished()—Called every time the loader finishes loading. Trigger an update of user-visible data in this method.
- onLoaderReset()—When the loader is reset, you usually want to invalidate the currently held data until new data has been loaded.

To implement these callbacks, you need to implement LoaderManager callbacks for the type of loader you have. For a cursor loader, change the signature of your activity as follows, then implement the callbacks.

```
public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<Cursor>
```

onCreateLoader()

This callback instantiates and returns a new loader instance of the desired type. Since the loader manager can be managing multiple loaders, an ID argument identifies the loader to instantiate. Once created, the loader will start loading data, and it will observe your date for changes, and reload as necessary.

To create CursorLoader, you need:

- uri—The URI for the content to retrieve from the content provider. This identifies the content provider and the data to observe to the loader.
- projection—A list of columns to return. Passing null will return all columns, which is inefficient.
- selection—A filter declaring which rows to return, formatted as a SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given URI.
- selectionArgs—You may include ?s in the selection, which will be replaced by the values from selectionArgs, in the order that they appear in the selection. The values will be bound as Strings.
- sortOrder — How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.

```
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    String queryUri = CONTENT_URI.toString();
    String[] projection = new String[] {CONTENT_PATH};
    return new CursorLoader(this, Uri.parse(queryUri),
            projection, null, null, null);
}
```

Notice how this is very similar to initiating a content resolver:

```
Cursor cursor = mContext.getContentResolver().query(Uri.parse(uri),
projection, selectionClause, selectionArgs, sortOrder);
```

onLoadFinished()

Specify what happens with the data once the loader has acquired it. In this function you should:

- Release the old data.
- Save the new data and, for example, make it available to your adapter.

The cursor loader monitors the data for you, so you do not, should not under any circumstances, do it yourself.

The loader also cleans up after itself, so there is no need for you to close the cursor.

If you are using a RecyclerView to display the data, all you need to do is hand the data over to the adapter whenever loading or reloading has finished.

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    mAdapter.setData(cursor);
}
```

onLoaderReset()

Called when a previously created loader is being reset, and thus making its data unavailable. You should clean all references to the data at this point. Again, if you are passing the data to an adapter for display in a RecyclerView, the adapter does the actual work, you just have to instruct it to do so.

```
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    mAdapter.setData(null);
}
```

Using the data returned by the loader

In the practicals, you are using a RecyclerView that is driven by an adapter to display the data fetched by the loader. Once the loader receives the data, it hands the data over to the adapter through, for example, a setData() call. The setData() method updates an instance variable in the adapter that holds the most current data set, and notifies the adapter that there is fresh data.

```
public void setData(Cursor cursor) {
    mCursor = cursor;
    notifyDataSetChanged();
}
```

The benefits of cursors

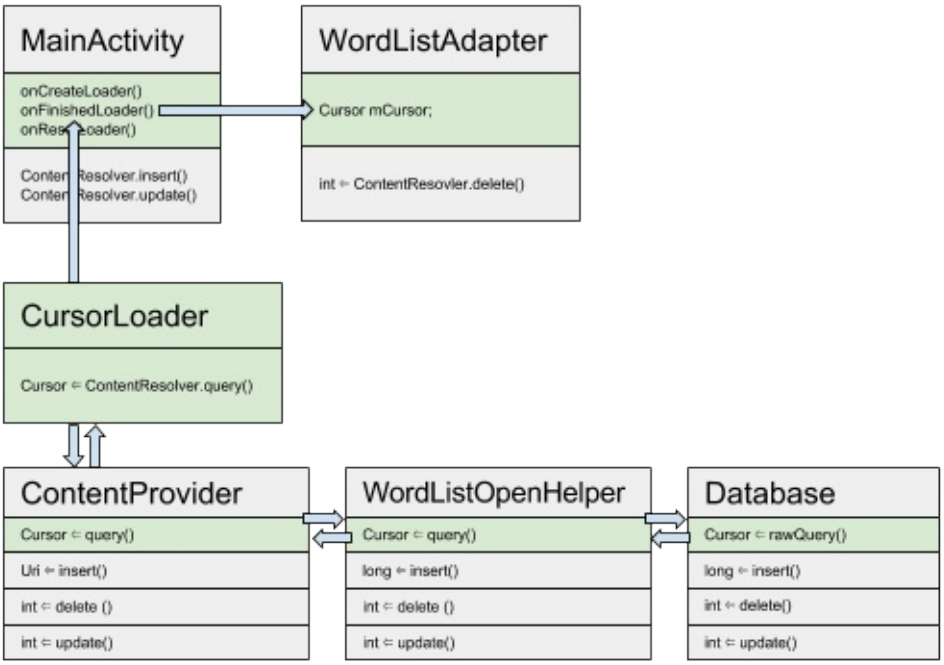
You may have noticed that the database uses cursors, the content provider uses cursors, and the loader uses cursors. Using the same data type throughout your backend, and only unpacking it in the adapter, where the contents of the cursor are prepared for display, makes for a uniform backend with clean interfaces. This makes it easier to write the code, easier to test, and easier to debug. It also makes the code simpler and shorter.

Complete app with methods

The following diagram shows the methods and data types that connect the different parts of an application that uses:

- A SQLite database to store data, and an SQLiteOpenHelper subclass to manage the database.
- A content provider to make data available to this (and other) apps.
- A loader to load data to display to the user.
- A RecyclerView.Adapter that displays and updates data shown to the user in a RecyclerView.

The green colored boxes show the call stack and journey of the cursor through the layers of the application for a query(). Note how inserting, deleting, and updating are still handled by the content resolver. However, the loader will notice any changes made by insert, delete, or update operations, and will reload the data as necessary.



Related practical

The related exercises and practical documentation is in Android Developer Fundamentals: Practicals.

- Load and Display Data Fetched from a Content Provider

Learn more

Developer Documentation:

- Loaders
- Running a query with a CursorLoader
- CursorLoader class

13.1: Permissions, Performance and Security

Contents:

- Permissions
- Performance
- Security best practices

You've now learned the fundamental core skills you need to build Android applications. This lesson discusses best practices as they relate to permissions, performance and security. This lesson does not have a corresponding practical.

Permissions

As you have worked through the practicals, there were times when your app needed to get permission to do something, including when it needed to:

- connect to the Internet.
- use a content provider in another app.

This section gives a brief overview of permissions so that you understand how and when your app needs to ask for permission before it can perform and action.

Ask permission if it isn't yours

An app is free to use any resources or data that it creates, but must get permission to use anything—data, resources, hardware, software—that does not belong to it. For example, your app must get permission to read the user's Contacts data, or to use the device's camera. It makes sense that an app needs permission to read a user's Contacts, but you might wonder why it needs permission to use the camera. It is because the camera hardware does not belong to the app, and your app must always get permission to use anything that is not part of the app itself.

Requesting permission

To request permission, add the `<uses-permission>` attribute to the Android manifest file, along with the name of the requested permission. For example, to get permission to use the camera:

```
<uses-permission android:name="android.permission.CAMERA"/>
```

Examples of permissions

The Android framework provides more than 100 predefined permissions. These include some obvious things including permission to access or write the user's personal data such as:

- reading and writing a user's Contacts list, calendar, or voicemail
- accessing the device's location
- accessing data from body sensors

Some of the other pre-defined permissions are less obvious, such as permission to collect battery statistics, permission to connect to the internet, and permission to use hardware such as the camera or fingerprint hardware.

Android includes pre-defined permissions for initiating a phone call without requiring the user to confirm it, reading the call log, capturing video output, rebooting the device, changing the date and time zone, and many more.

You can see all the system-defined permissions at https://developer.android.com/reference/android/Manifest.permission.html.

Normal and dangerous permissions

Android classifies permissions as normal or dangerous.

A **normal permission** is for actions that do not affect user privacy or user data, such as connecting to the Internet.

A **dangerous permission** is for an action that does affect user privacy or user data, such as permission to write to the user's voicemail.

Android automatically grants normal permissions but asks the user to explicitly grant dangerous permissions.

Note: Apps must list all permissions they use in the Android manifest, even normal permissions.

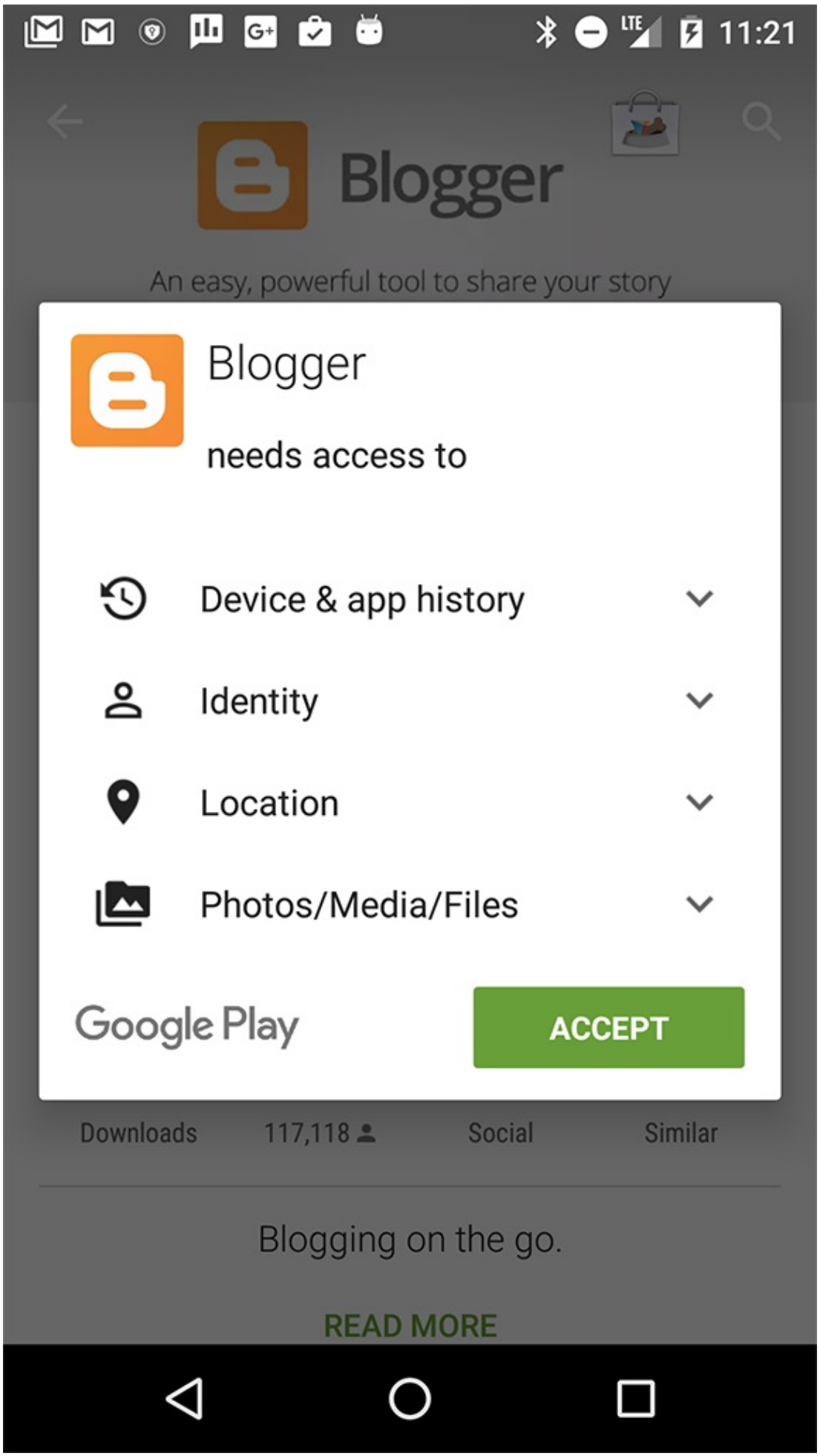
How users grant and revoke permissions

The way users grant and revoke permissions depends on:

- the version of Android that the device is running.
- the version of Android that the app was created for.

Before Marshmallow (Android 6.0)

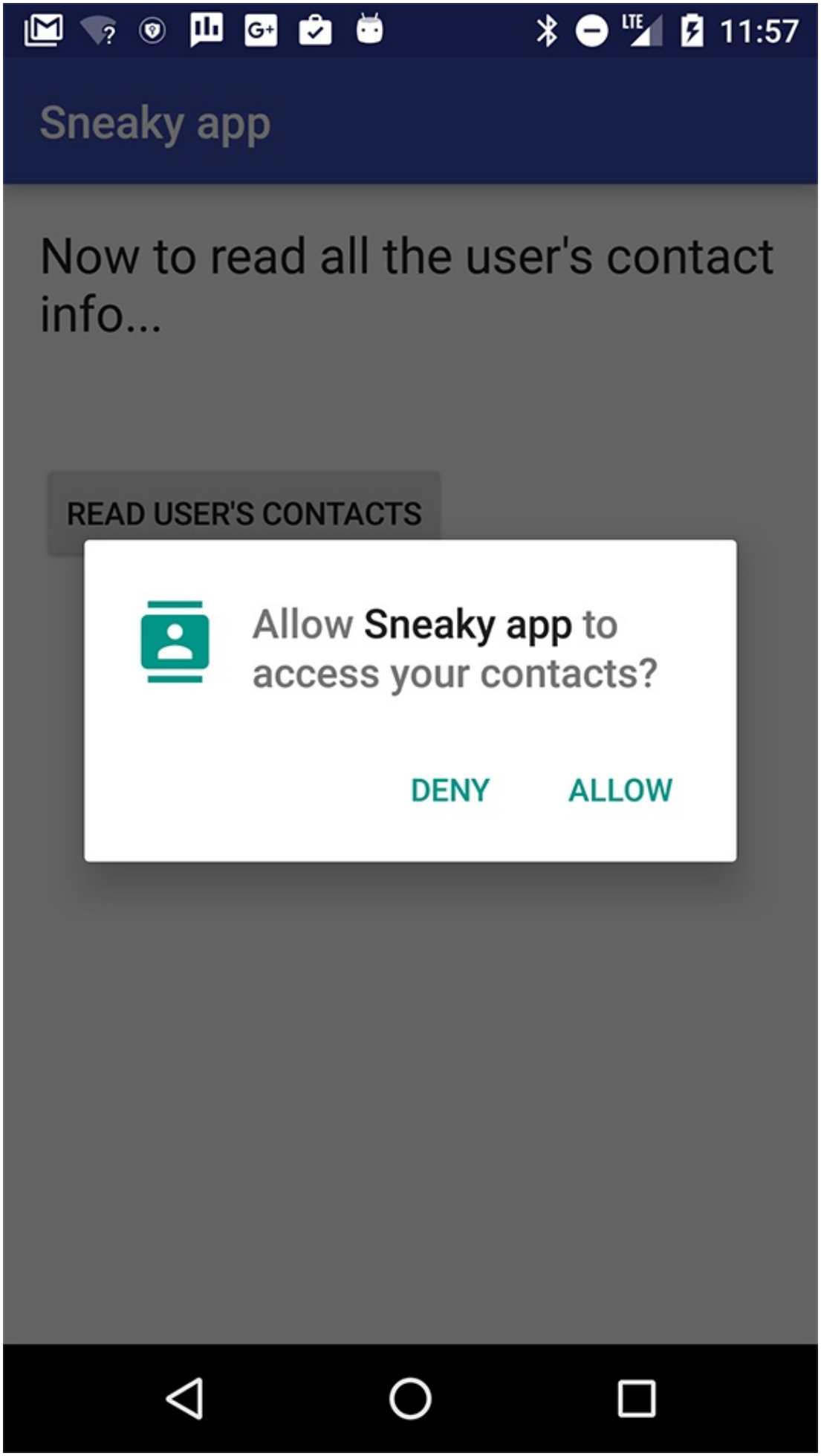
If an app was *created* for a version of Android before 6.0 (Marshmallow) ***or*** it is *running* on a device that uses a version of Android before Marshmallow, Google Play asks the user to grant required dangerous permissions ***before installing the app***.


11:21
Blogger
An easy, powerful tool to share your story
Blogger
needs access to
Device & app history
Identity
Location
Photos/Media/Files
Google Play
ACCEPT
Downloads
117,118
Social
Similar
Blogging on the go.
READ MORE

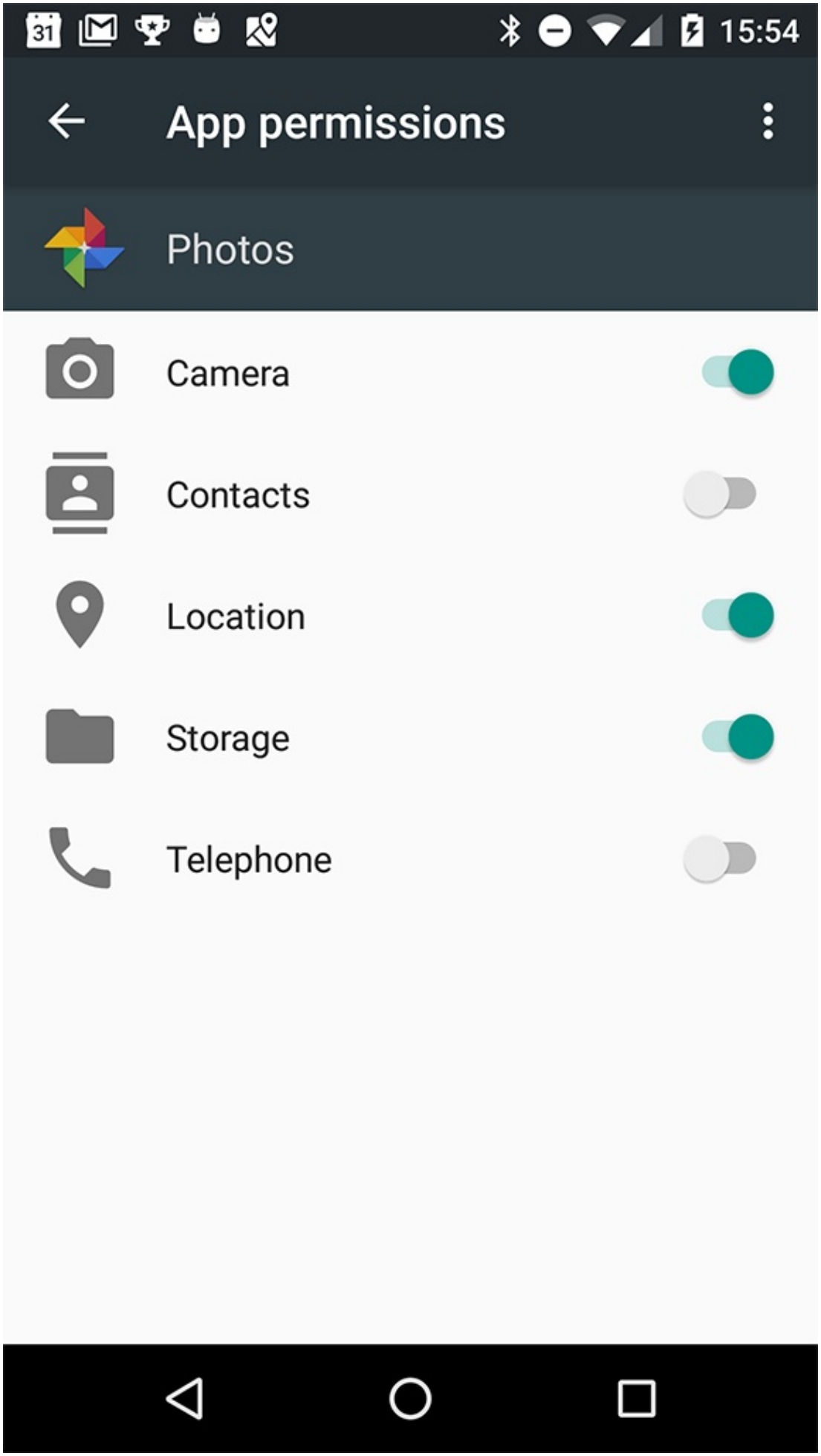
If the user changes their mind and wants to deny permissions to the app after it is installed, the only thing they can do is to uninstall the app.

Marshmallow onwards

If an app was created for a version of Android from Android 6.0 (Marshmallow) onwards ***and*** it is running on a device that uses a version of Android from Marshmallow onwards, then Google Play does not ask the user to grant dangerous permissions to the app before installing it. Instead, when the user starts to do something in the app that needs that level of permission, Android shows a dialog box asking the user to grant permission.


11:57
Sneaky app
Now to read all the user's contact info...
READ USER'S CONTACTS
Allow Sneaky app to access your contacts?
DENY
ALLOW

The user can grant or revoke individual permissions at any time. They do this by going to the Settings App, choosing Apps, and selecting the relevant app. In the Permissions section, they can enable or disable any of the permissions that the app uses.


15:54
App permissions
Photos
Camera
Contacts
Location
Storage
Telephone

How differences in the permissions models affect developers

In the "old" permissions model, Google Play and the Android Framework worked together to get permission from the user. All that the developer needed to do was to make sure that the app listed the permissions it needed in the Android manifest file. The developer could assume that if the app was running, then the user had granted permission. The developer did not need to write code to check if permission had been granted or not.

In the "new" permissions model, you can no longer assume that if the app is running, then the user has granted the needed permissions. The user could grant permission the first time they run the app, then, at any time, change their mind and revoke any or all of the permissions that the app needs.

So, the app must check whether it still has permission every time it does something that requires permission. The Android SDK includes APIs for checking if permission has been granted. Here is a code snippet that checks if the app has permission to write to the user's calendar:

```
// Assume thisActivity is the current activity
int permissionCheck = ContextCompat.checkSelfPermission(thisActivity,
   Manifest.permission.WRITE_CALENDAR);
```

The Android framework for Android 6.0 (API level 23) includes methods for checking for and requesting permissions. The Support Library also includes methods for checking for and requesting permission.

We recommend that you use the support library methods for handling permissions, because the permission methods in the support library take care of checking which version of Android your app is running on, and taking the appropriate action. For example, if the user's device is running an older version, then the `checkSelfPermission()` method in the support library checks if the user already granted permission at runtime, but if the device is running Marshmallow or later, then it checks if permission is still granted, and if not, shows the dialog to the user to ask for permission.

This lesson does not go into detail on how to use the APIs for handling permissions. See Requesting Permission at Runtime for details.

Best practices for permissions

When an app asks for too many permissions, users get suspicious. Make sure your app only requests permission for features and tasks it really needs, and make sure the user understands why they are needed.

Wherever possible, use an Intent instead of asking for permission to do it yourself. For example, if your app needs to use the camera, send an Intent to the camera app, and that way the camera app will do all your work for you and your app does not need to get permission to use the camera (and it will be much easier for you to write the code than if you accessed the camera APIs directly).

Learn more about Permissions in Android

- Pre-defined permissions
- Best practices for permissions
- Blog entry about runtime permissions

Performance

You have made your app as useful, interesting, and beautiful as possible. However, to make it stand out from the crowd, you should also make it as small, fast, and efficient as possible. Consider the impact your app might have on the device's battery, memory, and disc space. And most of all, be considerate of users' data-plans. The following recommendations are only the tip of the iceberg where performance is concerned, but they give you an idea on where to start.

Important: Maximizing performance is all about balance, finding and making the best trade-offs between app complexity, functionality, and visuals, to give users the best possible experience with your app.

Keep long-running tasks off the main thread

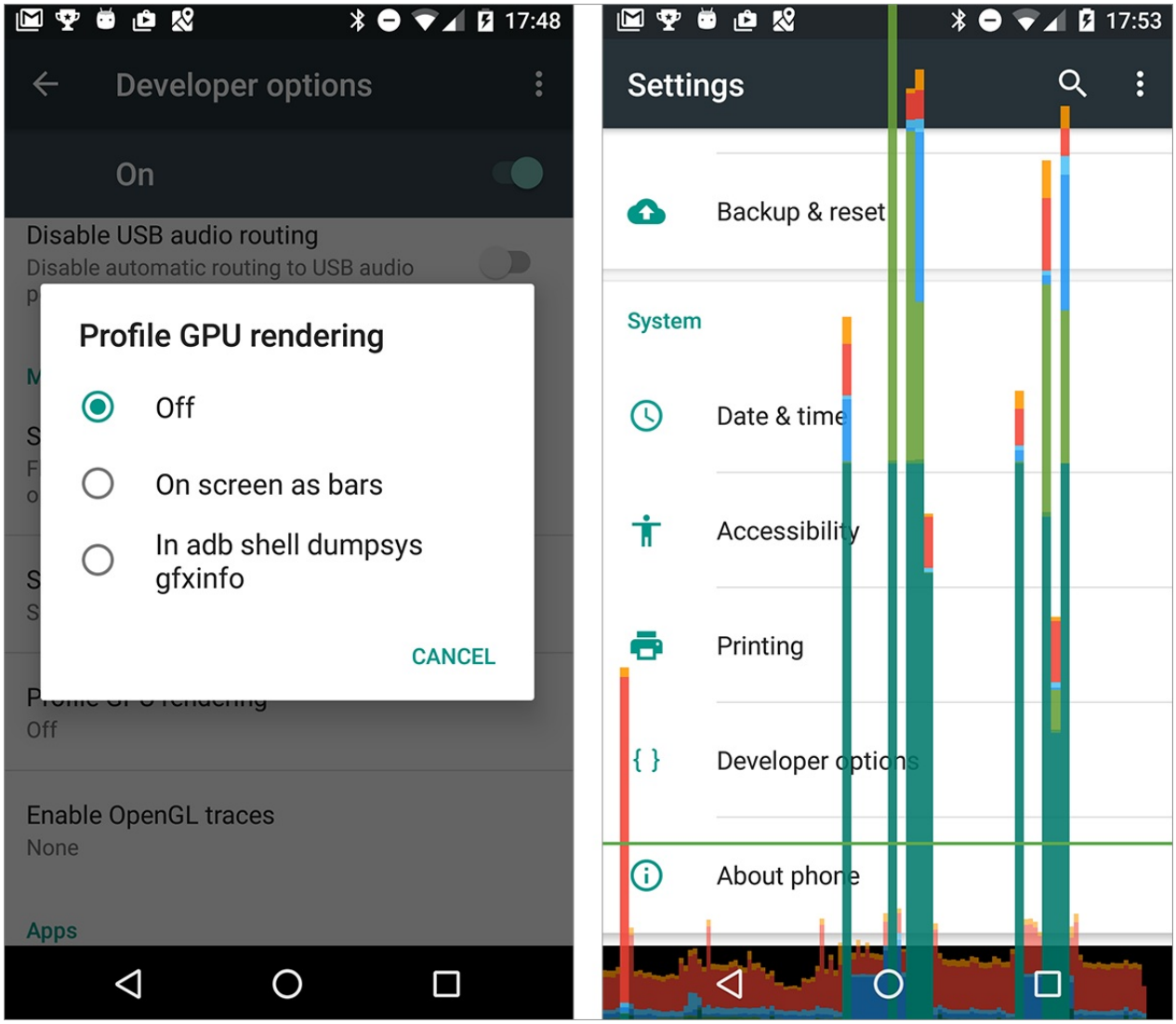
This course has already talked about moving work off the main thread into the background to help keep the UI smooth and responsive for the user. The hardware that renders the display to the screen typically updates the screen every 16 milliseconds, so if the main thread is doing work that takes longer than 16 milliseconds, the app might skip frames, stutter or hang, all of which are likely to annoy your users.

You can check how well your app does at rendering screens within the 16 millisecond limit by using the **Profile GPU Rendering** tool on your Android device.

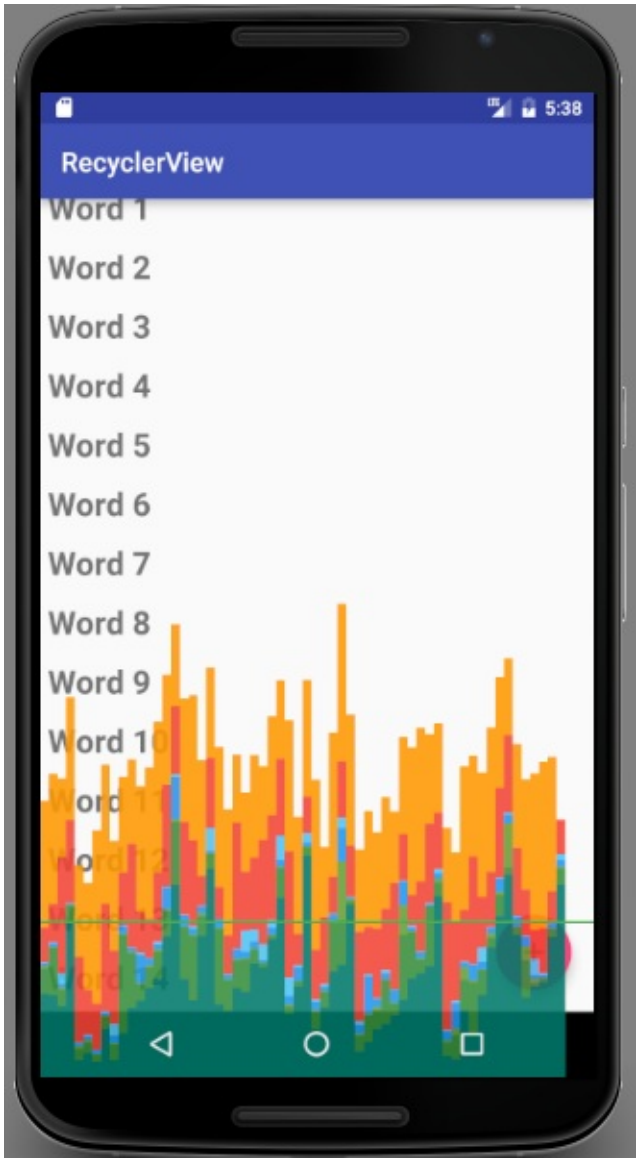
1. Go to **Settings** > **Developer options**.
2. Scroll down to the **Monitoring** section.
3. Select **Profile GPU rendering**.
4. Choose **On screen as bars** in the dialog box.

Immediately you will start seeing colored bars on your screen.

Note: You can run the tool on an emulator, just to try it, but the data is not indicative of how your app would perform on a real device.



Open your own app, and watch the colored bars.



One bar represents one screen rendered. If a bar goes above the green line, it took more than 16 ms to render. The colors in the bar represent the different stages in rendering the screen.

Read about how to interpret the results and what the different stages mean at Analyzing with Profile GPU Rendering. If you spend time using the Profile GPU rendering tool on your app, it will help you identify which parts of the UI interaction are slower than might be expected, and then you can take action to improve the speed of your app's UI.

For example, if the green Input portion of the bar is large, your app spends a lot of time handling input events, that is, executing code called as a result of input event callbacks. To fix this, consider when and how you request user input, and whether you can handle it more efficiently.

Simplify your UI

This course has talked about how to make your apps interesting and visually compelling using material design guidelines, and it's taught you how to use the Layout Editor to create your layouts. You've learned that you can create nested hierarchies of layouts. You've learned how to use drawables as background elements for your views. These elements allow you to create complex nested layouts with diverse backgrounds and views overlapping each other throughout your app.

However, your layouts will draw faster and use less power and battery if you spend time designing them in the most efficient way.

Try to avoid:

- Deeply nested layouts—If your layouts are narrow and deep, the Android system has to perform more passes to lay out all the views than if your view hierarchy is wide and shallow. Consider how you can combine, flatten, or even eliminate views.
- Overlapping views—this results in "overdraw" where the app wastes time drawing the same pixel multiple times, and only the final rendition is visible to the user. Consider how you can size and organize your views so that every pixel is only drawn once or twice.

Simplify layouts

Make sure your layouts include only the views and functionality your app needs. Simple layouts are generally more visually appealing to users, and they draw faster, giving you a double win.

Flatten your layouts as much possible, which means reducing the number of nested levels in your app's view hierarchy. For example, if your layout contains a LinearLayout inside a LinearLayout inside a LinearLayout, you may be able to arrange all the views inside a single ConstraintLayout.

See the guide Optimizing your UI for more information on improving the performance of your app's UI.

Minimize overlapping views

Imagine that you are painting the door of your house in red. They you paint it again in green. Then you paint it again in blue. In the end, the only color you see is blue, but you wasted a lot of energy painting the door multiple times.

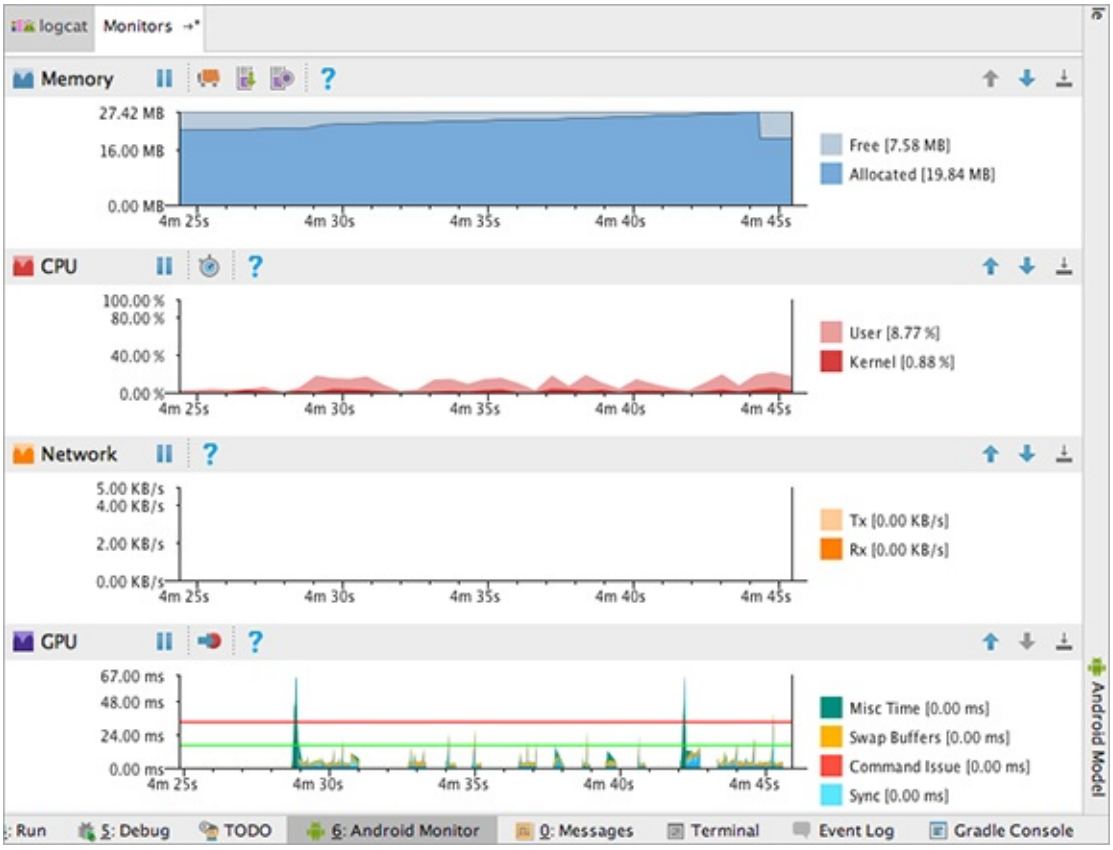
Each layout in your app is like the door. Every time your app "paints" (draws) a pixel, it takes time. If your layout has overlapping views, then your app is using time and resources drawing pixels that it then draws over again. Try to reduce the amount of times your app overdraws pixels, by reducing overlapping views. Be careful about using drawable backgrounds on overlapping views and only use them when they are visible.

See the guide Reducing Overdraw for more information.

Monitor the performance of your running app

Android Studio has tools to measure your app's memory usage, GPU, CPU, and Network performance. App crashes are often related to memory leaks, which is when your app allocates memory and does not release it. If your app leaks memory, or uses more memory than the devices makes available, it will eventually use up all the available memory on the device. Use the Memory Monitor tool that comes with Android Studio to observe how your app uses memory.

1. In Android Studio, at the bottom of the window, click the **Android Monitor** tab. By default this opens on logcat.
2. Click the **Monitors** tab next to logcat. Scroll or make the window larger to see all four monitors: Memory, CPU, Networking, and GPU.
3. Run your app and interact with it. The monitors update to reflect the app's use of resources. Note that to get accurate data, you should do this on a physical, not virtual, device.



The monitors are:

- **Memory monitor**—Reports how your app allocates memory and helps you to visualize the memory your app uses.
- **CPU monitor**—Lets you monitor the central processing unit (CPU) usage of your app. It displays CPU usage in real time.
- **GPU monitor**—Gives a visual representation of how long the graphical processing unit (GPU) takes to render frames to the screen.
- **Network Monitor**—Shows when your application is making network requests. It lets you see how and when your app transfers data, and optimize the underlying code appropriately.

Read the Android Monitor page to learn more about using the monitors.

Learn more about improving your app's performance

- You, Your App, and Performance
- Exceed the Android speed limit

Overdraw:

- Only draw what you see
- Reducing Overdraw

Tools:

- Performance profiling tools
- Optimizing your UI
- Analyzing with profile GPU rendering
- Android Monitor

Security best practices

Much of the burden of building secure apps is handled for you by the Android Framework. For example, apps are isolated from each other so they can't access each other or use each other's data without permission.

However, as an app developer, you have the responsibility to make sure your app treats the user's data safely and with integrity. Your app is also responsible for keeping its own data safe.

Handling user data

This lesson has already discussed how Android uses permissions to make sure apps cannot access the user's personal data without their permission. But even if the user gives your app permission to access their private data, do not do so unless absolutely necessary. And if you do, treat the data with integrity and respect. For example, just because the user gives your app permission to update their calendar does not mean you have permission to delete all their calendar entries.

Android apps operate on a foundation of implied trust. The users trust that the apps will use their data in a way that makes sense within the context of the app.

If your app is a messaging app, it's likely that the user will grant it permission to read their contacts. That does not mean your app is allowed to read all the user's contacts and send everyone a spam message.

Your app must only read and write the user's data when absolutely necessary, and only in a way that the user would expect the app to do so. Once your app has read any private data, you must keep it safe and prevent any leakage. Do not share private data with other apps.

Depending on how your app uses user data, you might also need to provide a written statement regarding privacy practices when you publish your app in the Google Play store.

Be aware that any data that the user acquires, downloads, or buys in your app belongs to them, and your app must store it in a way that the user still has access to it even if they uninstall your app.

Important: Logs are a shared resource across all apps. Any app that has the READ_LOGS permission can read all the logs. Do **not** write any of the user's private data to the logs.

Public Wi-Fi

Many people use mobile apps over public Wi-Fi. When did you last access the Internet from your mobile phone over the public Wi-Fi at a coffee shop, an airport, or a railway station?

Design your app to protect your user's data when they are connected on public Wi-Fi. Use `https` rather than `http` whenever possible to connect to websites. Encrypt any user data that gets transmitted, even data that might seem innocent like their name.

For transmitting sensitive data, implement authenticated, encrypted socket-level communication using the SSLSocket class. This class adds a layer of security protections over the underlying network transport protocol. Those protections include protection against modifications of messages by a wiretapper, enhanced authentication with the server, and increased privacy protection.

Validating user input

If your app accepts input (and almost every app does!) you need to make sure that the input does not bring anything harmful in with it.

If your app uses native code, reads data from files, receives data over the network, or receives data from any external source, it has the potential to introduce a security issue. The most common problems are buffer overflows, dangling pointers, and off-by-one errors. Android provides a number of technologies that reduce the exploitability of these errors, but they do not solve the underlying problem. You can prevent these vulnerabilities by carefully handling pointers and managing buffers.

If your app allows users to enter queries that are submitted to an SQL database or a content provider, you must guard against SQL injection. This is a technique where malicious users can inject SQL commands into a SQL statement by entering data in a field. Injected SQL commands can alter SQL statement and compromise the security of the application and the database.

Read about using parameterized queries to defend against SQL injections in the content providers section of the Security Tips guide.

Another thing you can do to limit the risk of SQL injection is to use or grant either the READ_ONLY or WRITE_ONLY permission for content providers.

WebViews

One of the View classes in Android is WebView, which displays a web page.

This course has not discussed WebView, but you might have found it and tried it out for yourself. We are mentioning it here because even though it is very cool to quickly display a web page in your app, WebView does come with security concerns.

Because WebView consumes web content that can include HTML and JavaScript, it can introduce common web security issues such as cross-site-scripting (JavaScript injection).

By default, a WebView provides no browser-like widgets, does not enable JavaScript, and ignores web page errors. If your goal is only to display some HTML as a part of your UI, this is acceptable as the user won't need to interact with the web page beyond reading it, and the web page won't need to interact with the user. If you want a fully functional web browser, invoke the Browser application with a URL Intent rather than showing the page with a WebView. This is also a more secure option than extending the WebView class and enabling features such JavaScript.

Security, like performance, is a large topic that cannot be covered in a few paragraphs. It is your responsibility to treat user data with care and keep it safe at all times. Use the resources below to learn as much as you can about treating your users and their data with the highest regard.

Learn more about security best practices

- Security tips
- Input validation
- Security tips for content providers

14.1: Firebase and AdMob

Contents:

- Firebase
  - Get started with Firebase
  - Firebase Analytics
  - Firebase Notifications
  - Firebase Realtime Database
  - Firebase Test Lab
  - Firebase Demo
  - More Firebase features
  - Learn more about Firebase
- Make money from your app
- AdMob
  - Create an AdMob account
  - Implement AdMob in your app
  - Learn more about AdMob

This lesson covers three topics:

- Firebase, a set of tools for mobile and web app developers
- Making money from your Android app
- Running ads using AdMob

Firebase

Firebase is a set of tools for app developers, but not just for Android app developers. It is for iOS app developers and web app developers too. However, since this is a course about Android development, this lesson only talks about how to use Firebase with Android apps.

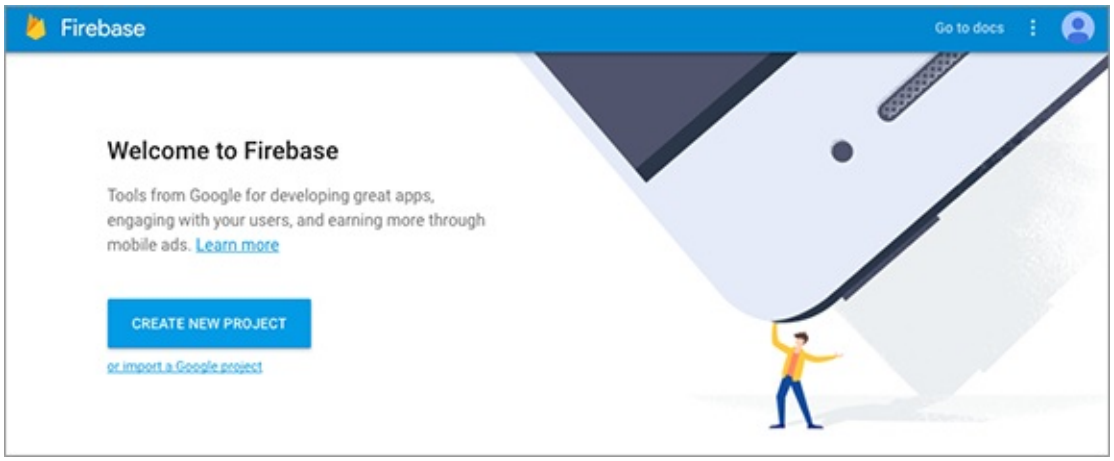
As an Android developer, you use Android Studio to build your app, but you can use Firebase to add features to your app, get a wider audience for your app, test your app, earn revenue from your app, and get analytics on the usage of your app.

This chapter does not discuss everything about Firebase, but it introduces Firebase, helps you get started using it, and highlights important features that you might want to use.

Get started with Firebase

To use Firebase, go to the Firebase console at https://console.firebase.google.com/. You will need to have a Google account.

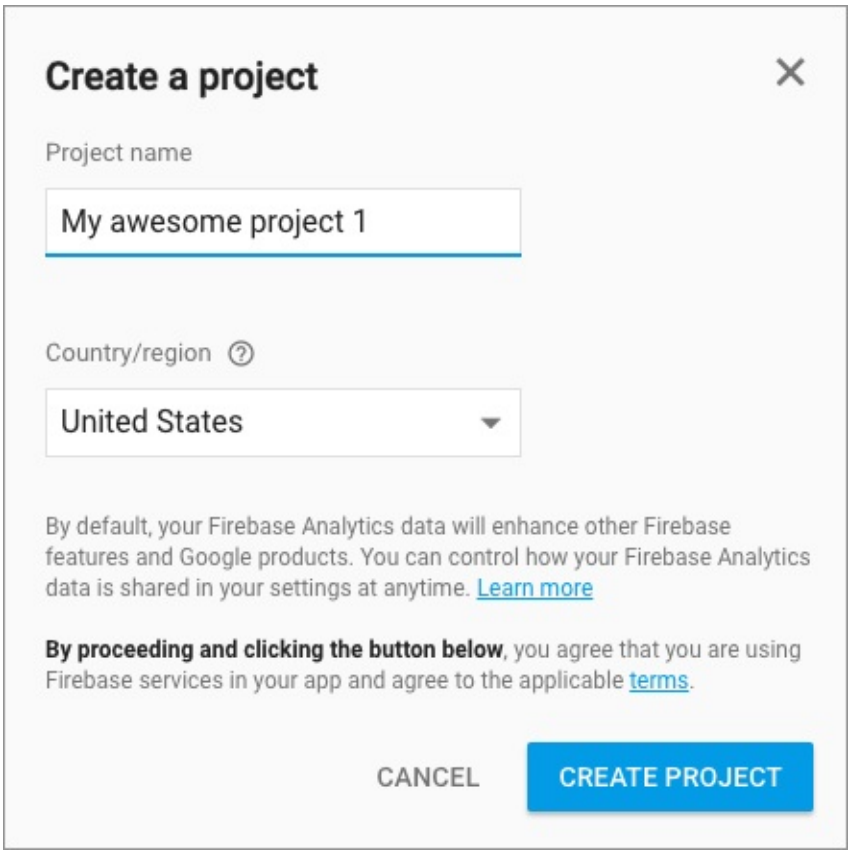
The first time you go to the Firebase you see a welcome screen that includes a button to create a new project.


Firebase
Go to docs
Welcome to Firebase
Tools from Google for developing great apps, engaging with your users, and earning more through mobile ads. Learn more
CREATE NEW PROJECT
or import a Google project

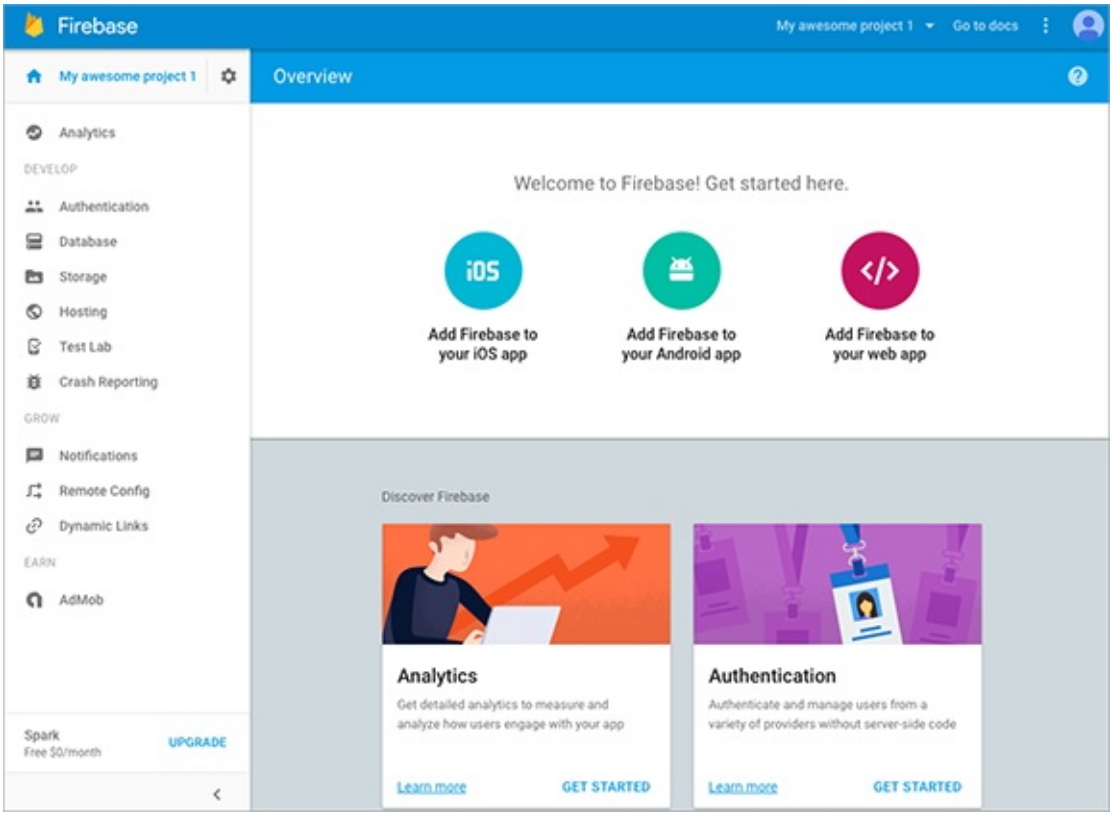
To use Firebase features with your Android app, first create a Firebase project and then add your Android app to the Firebase project.

A project is a container for your apps across platforms: Android, iOS, and web. You can name your project anything you want; it does not have to match the name of any apps.

1. In the Firebase console, click the **CREATE NEW PROJECT** button.
2. Enter your Firebase project name in the dialog box that appears then click **Create Project.**


Create a project
Project name
My awesome project 1
Country/region
United States
By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more
By proceeding and clicking the button below, you agree that you are using Firebase services in your app and agree to the applicable terms.
CANCEL
CREATE PROJECT

3. The Firebase console opens. Look in the menu bar for the name of your project.



Add your Android app to your Firebase project

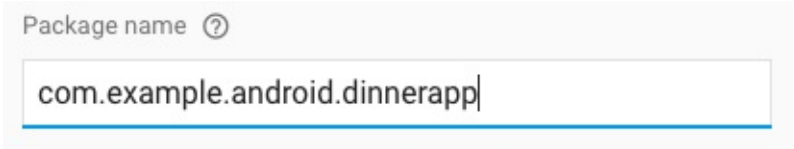
The next step is to associate your Android application with your Firebase project. First though, get the information you will need. To connect Firebase to your Android app, you need to know the package name used in your app. It's best to have your app open in Android Studio before you start the process.

To connect your Firebase project and your Android app to each other:

1. In Android Studio, open your Android app.
2. Make sure you have the latest Google Play services installed.

   Tools > Android SDK Manager > SDK Tools tab > Google Play services.

3. Note the package of the source code in your Android app.
4. In the Firebase console, Click **Add Firebase to your Android app**.
5. Enter the package name of your app in In the dialog box that appears.



6. Click **Add App**. Firebase downloads a file called `google_services.json` to your computer. Save it in a convenient location, such as your Desktop. (If you have any trouble, click Project Settings (the cogwheel next to Overview) and download the file manually.)
7. Find the downloaded file on your computer.
8. Copy or move that file into the **app** folder of your project in Android Studio. The wizard in Firebase shows where to put the file in Android Studio.
9. In the Firebase console, click **Continue**. The screen now shows the instructions for updating your build.gradle files.
10. In Android Studio, update the project-level build.gradle (Project: app) file with the latest version of the google-services package (where x.x.x are the numbers for the latest version. See the Android setup guide for the latest version.)

```
buildscript {
  dependencies {
    // Add this line
    classpath 'com.google.gms:google-services:x.x.x'
  }
}
```

11. In Android Studio, in the app-level build.gradle (Module: app) file, at the bottom, apply the google-services plugin.

    `// Add to the bottom of the file apply plugin: 'com.google.gms.google-services'`

12. In Android Studio, sync the Gradle files.

13. In the Firebase console, click **Finish**.

Your Android app and your Firebase project are now connected to each other.

Firebase Analytics

You can enable Firebase Analytics in your app to see data about how and where your app is used. You will be able to see data such as how many people use your app over time and where in the world they use it.

All the data that your app sends to Firebase is anonymized, so you never see the actual identity of *who* used your app.

To get usage data about your app, add the `firebase-core` library to your app as follows:

1. Make sure you have added your app to your Firebase project.
2. In Android Studio, make sure you have the latest Google Play services installed.

   Tools > Android SDK Manager > SDK Tools tab > Google Play services

3. Add the dependency for firebase-core to your app-level build.gradle (Module: app) file:

   `compile 'com.google.firebase:firebase-core:x.x.x'`

After you add the `firebase-core` library to your app, it will automatically send usage data when people use your app. Without having to add any more code, you will get a default set of data about how, when, and where people use your app. Not only do you not have to add any code to generate default usage data, you don't even have to publish your app or do anything else to it other than use it.

When someone does something in your app, such as click a button or go to another activity, the app will *log an Analytics event*. This means that the app will package up the data about the event that happened, and put it in the queue to send to Firebase.

Check to see if the analytics event occurs

Android sends Analytics events in batches to minimize battery and network usage, as explained in this blog post. Generally speaking, Analytics events are sent approximately every hour or so to the server, but it also takes additional time for the Analytics servers to process the data and make it available to reports in the Firebase console.

While you are developing and testing your app, you don't have to wait for the data to show up in the Analytics dashboard to check that your app is logging Analytics events. As you use your app, make sure your logcat is displaying "Debug" messages. Look in the log for statements such as:

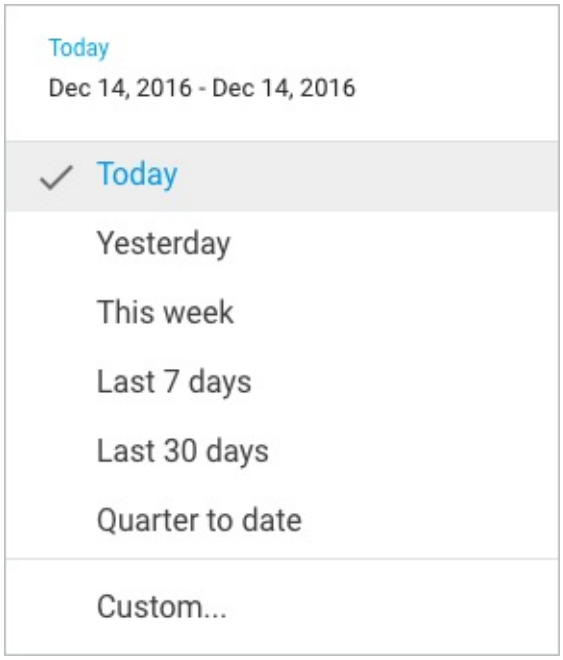
```
Logging event (FE): _e, Bundle[{_o=auto, _et=5388, _sc=SecondActivity, ...}]
```

These kinds of log messages indicate that an Analytics event has been generated. In this example, an Analytics event was generated when the `SecondActivity` started.

View reports in the Firebase Analytics dashboard

To see the Analytics reports, open the Firebase Console and select **Analytics**. The dashboard opens showing reports for your app for the last 30 days.

Note: The end date is always **Yesterday** by default. To see the usage statistics including **Today**, change the default date range to **Today**. Look for the calendar icon towards the top right of the screen and change the date range.



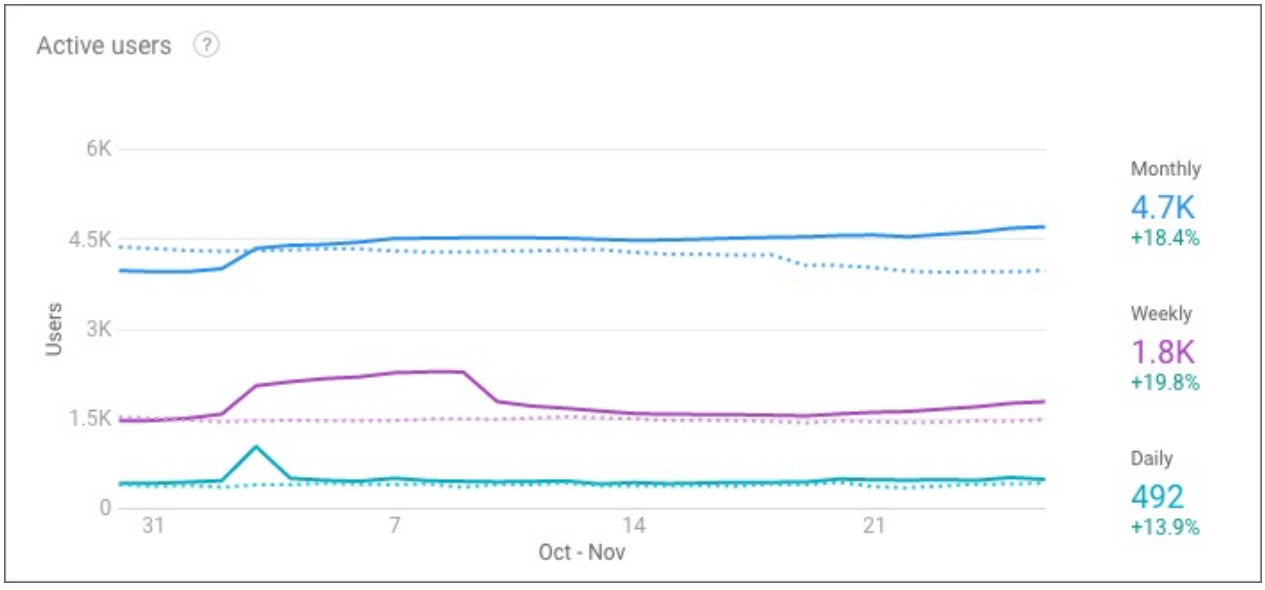
Default Analytics

The analytics information you get by default includes:

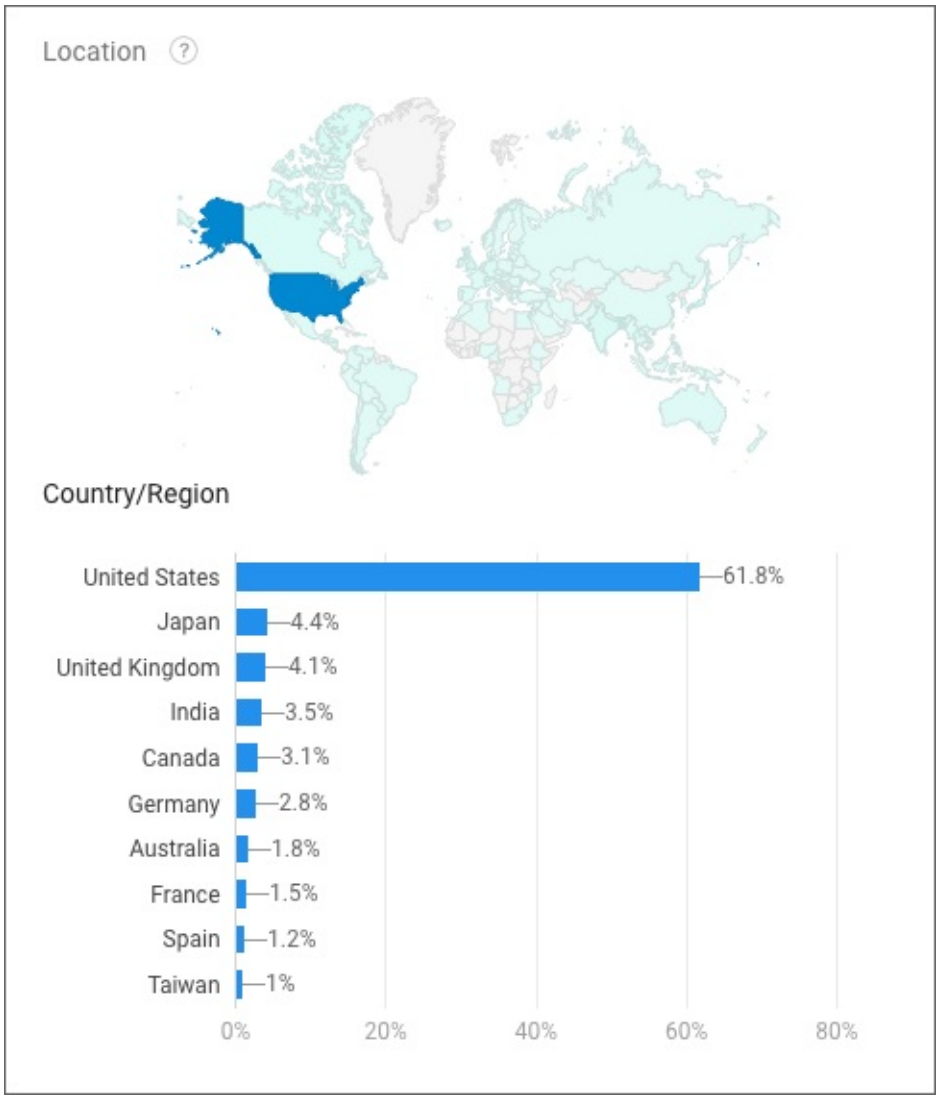
- Number of users
- Devices your users used
- Location of the users
- Demographics such as gender
- App version
- User engagement
- And more

For a full list of the available reports see the Firebase help for the Dashboard.

Here's an example of a report showing the number of people who used an app in the past 30 days:



And here's an example of a report showing the users' locations:



Other kinds of Analytics data

To see usage data for your app beyond the default reports, you need to add code to your app to send an "Analytics event" at the appropriate point in your app.

- In your app's main activity, define a variable for the instance of `FirebaseAnalytics` for your app.

  ```
  FirebaseAnalytics mFirebaseAnalytics;
  ```

- In the main activity's `onCreate()`, get the instance of `FirebaseAnalytics`.

  ```
  mFirebaseAnalytics = FirebaseAnalytics.getInstance(this);
  ```

- To send usage data in your app, call `logEvent()` on the `FireBaseAnalytics` instance.

You can log Analytics events either for pre-defined events or for your own custom events. The predefined events include:

- ADD_PAYMENT_INFO
- ADD_TO_CART
- LEVEL_UP
- LOGIN
- SIGN_UP

and many more.

For example, to send an Analytics event when a user goes to the next level in your app, you could use code such as:

```
mFirebaseAnalytics.logEvent(LEVEL_UP, null);
```

The second argument is a `Bundle` that contains information describing the event.

You can explore the predefined events and parameters in the `FirebaseAnalytics.Event` and `FirebaseAnalytics.Param` reference documentation.

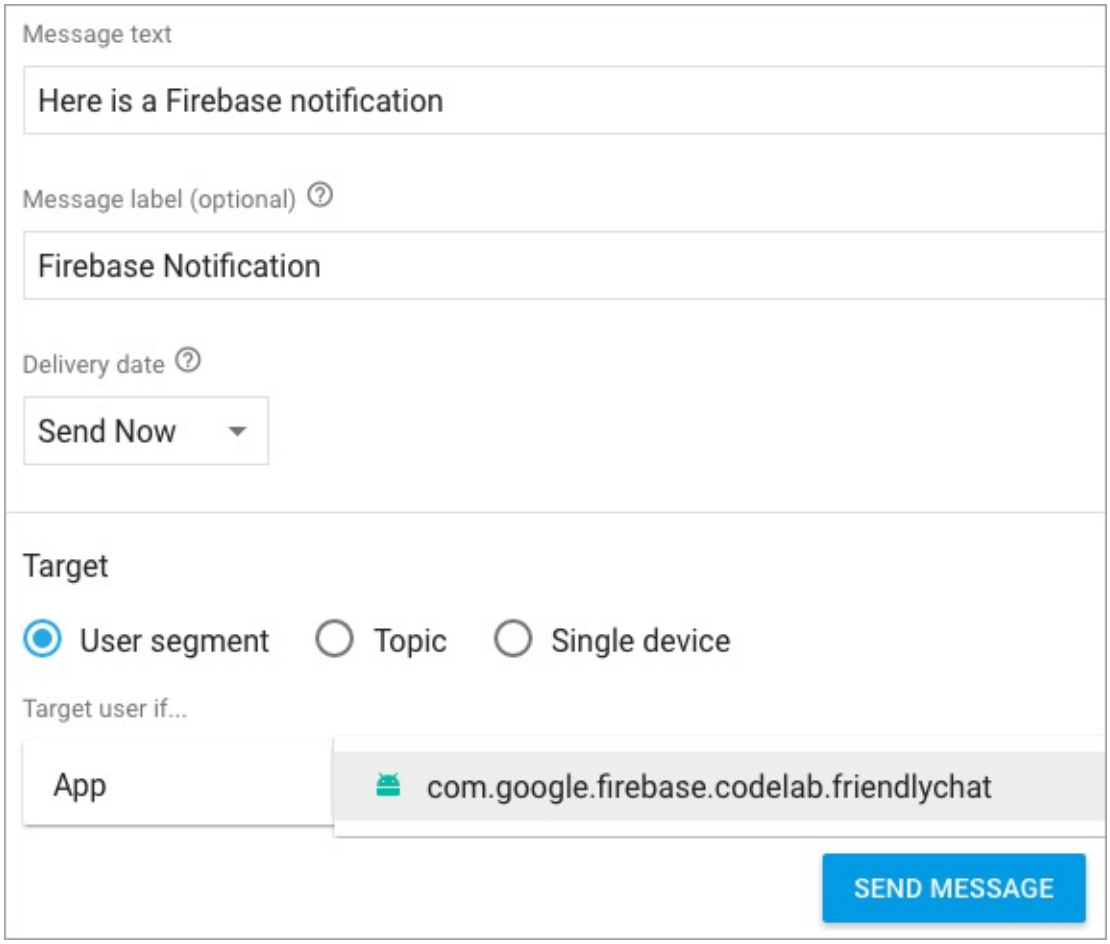
Read more about logging Analytics events in the Logging Analytics Events guide.

Firebase Notifications

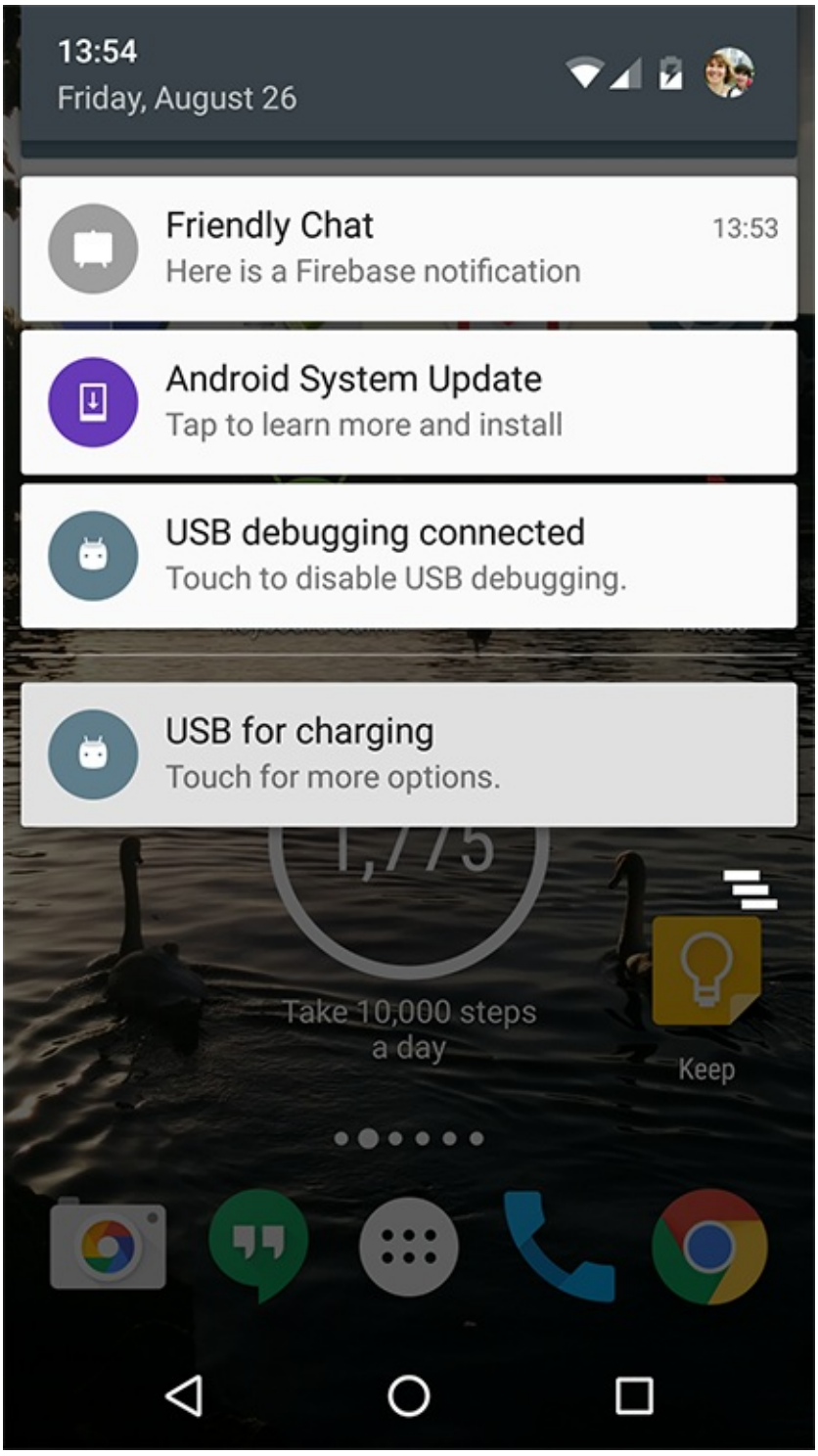
You learned in a previous lesson how to enable *your app* to send notifications to the user. Using the Firebase console, *you* can send notifications to all your users, or to a subset of your users.

Type the message in the Notifications section of the console, select the the audience segment to send the message to, and then send the message.

To send the message to all users of your app, set the **User Segment** to **App**, and select the package for your app.



Behind the scenes, Firebase uses Firebase Cloud Messaging to send the notification to the targeted devices where the selected app is installed.



Best practices for sending notifications

The same kind of best practices are true for sending notifications from the Firebase console as they are for sending notifications from the app. Be considerate of the user. Send only notifications that are important. Do not irritate your users by sending too many notifications or notifications with unhelpful or annoying content.

Firebase Realtime Database

In this course, you learned the different ways your app can save data. You used `SharedPreferences` to save data as key-value pairs, and you saved data in Android's built-in SQLite database. You created a content provider to provide an interface to access stored data, regardless of how it is stored, and also to share data with other apps.

But how do you share data across multiple clients such as different devices and apps, including Android, iOS and web apps, and allow them to update the data and all stay synchronized with the data in realtime? For this, you need a central cloud-based data repository.

Firebase offers a database that provides cloud-based data storage, allowing clients to all stay in sync as the data changes. If an app goes offline, the data remains available. When the app reconnects to the Internet, the data is synced to the latest state of the database.

For more information, see the Firebase database guide at:

- firebase.google.com/docs/database/

How data is stored

The Firebase Realtime Database is a NoSQL database. Data is stored as JSON objects. (You used JSON in the lesson about connecting to the Internet.)

You can think of the database as a cloud-hosted JSON tree. Unlike a SQL database, there are no tables or records. When you add data to the JSON tree, it becomes a node in the existing JSON structure with an associated key.

Here's an example of a JSON tree of data storing data about books and movies:

```
{
  "books": {
    "book one": {
      "title": "How to develop Android apps",
      "author": "Jane Developer"
    },
    "book two": {
      "title": "How to use Firebase",
      "author": "Adam Writer"
      },
    "book three": {
      "title": "How to search Google",
      "author": "Ava Searcher"
    }
  },
  "movies": {
    "movie one": {
      "title": "Saving the world",
      "main role": "Super man"
    },
    "movie two": {
      "title": "Saving the moon",
      "main role": "Bat girl"
      },
    "movie three": {
      "title": "Saving Mars",
      "main role": "Martian dog"
    }
   }
}
```

Here's an example of how this data appears in the Firebase Database console:

When you write code to access data in the database, you retrieve data items by their location. The location of the data is constructed by traversing the tree.

For example, the location to:

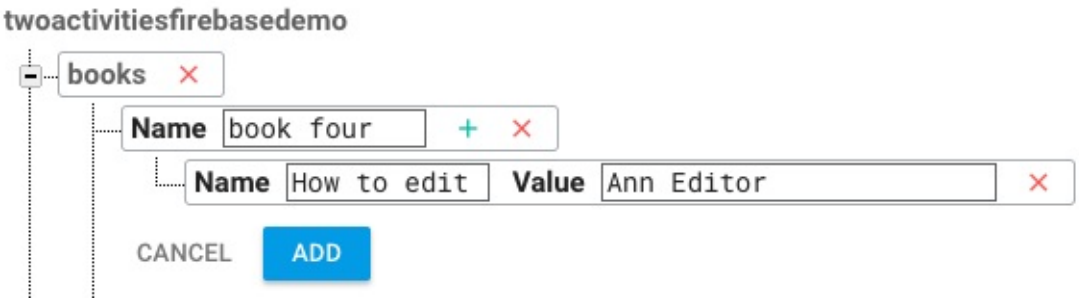
- **books** node is twoactivitiesfirebasedemo/database/data/books
- **book one** is twoactivitiesfirebasedemo/database/data/books/book%20one
- **author of book one** is twoactivitiesfirebasedemo/database/data/books/book%20one/author

Get and set data

You can use the Firebase console to view and update data in the database. *Your app* can use API calls to get and set data in the database.

Viewing and writing data in the Firebase console

In the Firebase console, you can add, edit and delete data.



You can also import and export data in the console. To import data, you must format the data as JSON and save it in a file with a `.json` extension.

Be aware that if you import a JSON file, it **overwrites** any data nodes that are defined in the file that already exist in the database.

Reading and writing data in your app

To use a Firebase database in your app, add the `firebase-database` library to your app, and then use methods on `DatabaseReference`:

1. Make sure you have added your app to your Firebase project.
2. Add the dependency for firebase-database to your app-level build.gradle (Module:app) file (where x is the latest version, see the Android Firebase Database developer guide for the latest version): `compile 'com.google.firebase:firebase-database:x.x.x'`
3. To access the data from your Android app, use the methods in the `DatabaseReference` class.

The easiest way to construct and handle the data is to create a Java object that represents that data. For example, you could create the `Book` class:

```
public class Book {

    public String title;
    public String author;

    public Book() {
        // Default constructor is required
    }

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

}
```

To access the data from your Android app, use the methods in the `DatabaseReference` class to access, create, and update data. The `DatabaseReference` class represents a reference to a data item.

For example:

- `child()` accesses a child node.
- `setValue()` sets the data at a node, and creates the node if it does not already exist.

To create a new data node:

1. Get the reference to the parent of the new data node
2. Call `child()` to create a reference to the new child node.
3. Use `setValue()` to set the value of the new node.

For example, suppose that you want to add a new book to the database with the following details:

- bookId = "book 4"
- title = "How to edit data"
- author = "Ann Editor"

To add this new book to the example data set shown previously, create an instance of `DatabaseReference`, and initialize it in the main activity's `onCreate()` method:

```
private DatabaseReference mDbBooksRef;
@Override
protected void onCreate(Bundle savedInstanceState) {
  // Create a new reference to the "books" node
  mDbBooksRef = FirebaseDatabase.getInstance().getReference("books");
  // rest of onCreate() …
}
```

Define a method to create a new `Book` object and add it to the database under the "books" node:

```
private void addNewBook(String bookId, String title, String author) {
    // Create a new Book object with the relevant details
    Book book = new Book(title, author);
    // mDbBooksRef is a reference to the "books" node
    // Create a child node whose key is bookId
    // and set its value to the new book
    mDbBooksRef.child(bookId).setValue(book);
}
```

Invoke the `addBook()` method as follows to create the new data item in the database:

```
addNewBook("book four", "How to edit data", "Ann Editor");
```

To learn more about reading and writing Firebase data in your app, see:

- Set up Firebase Realtime Database for Android
- Read and Write Data on Android.

Access control

Firebase provides a set of rules to determine who is allowed to view and update data in a Firebase database. Learn how to create the rules in Get Started with Database rules.

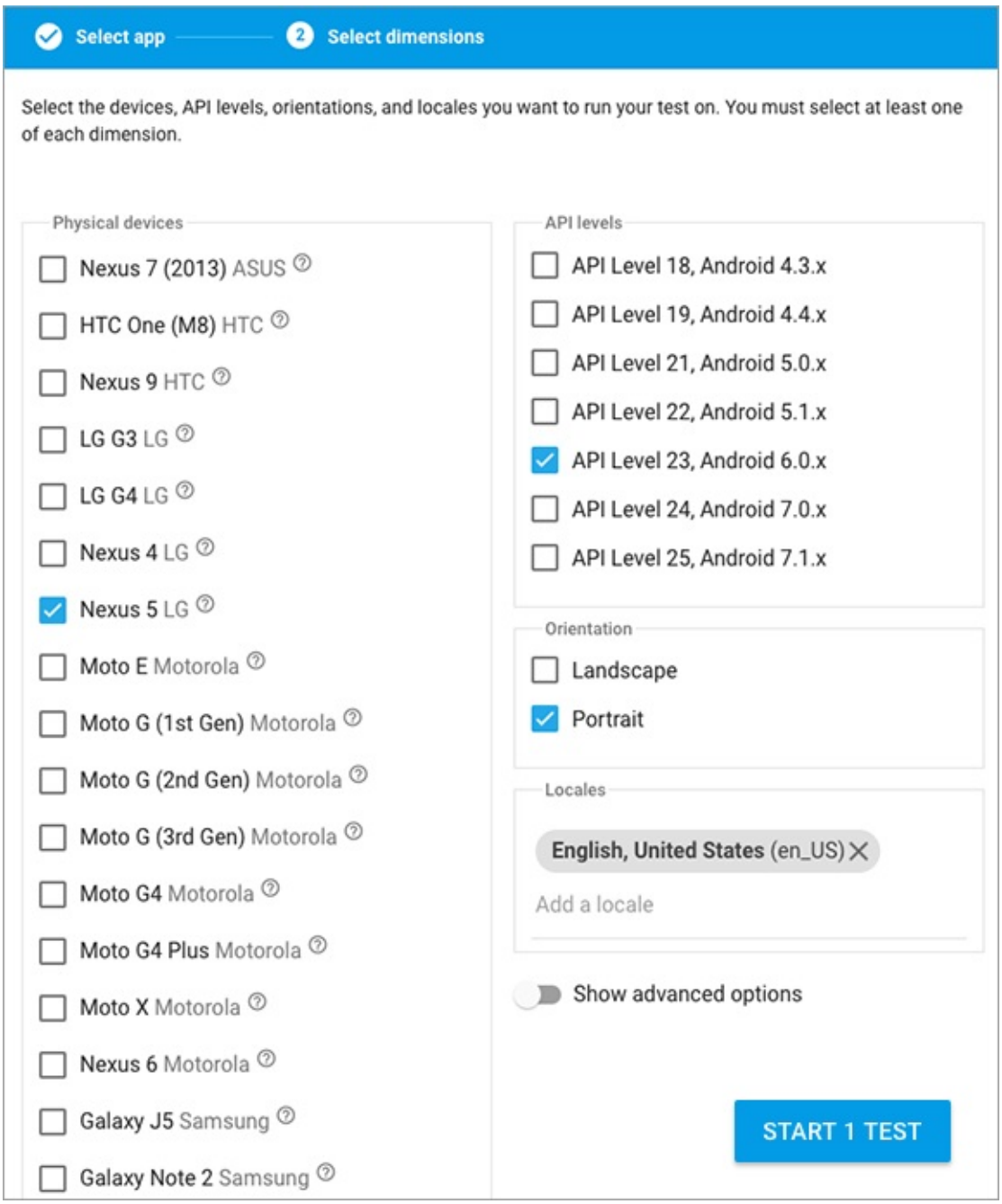
Firebase Test Lab

The Firebase Test Lab lets you test your app on a range of different devices hosted in Google's data centers and get reports on the results.

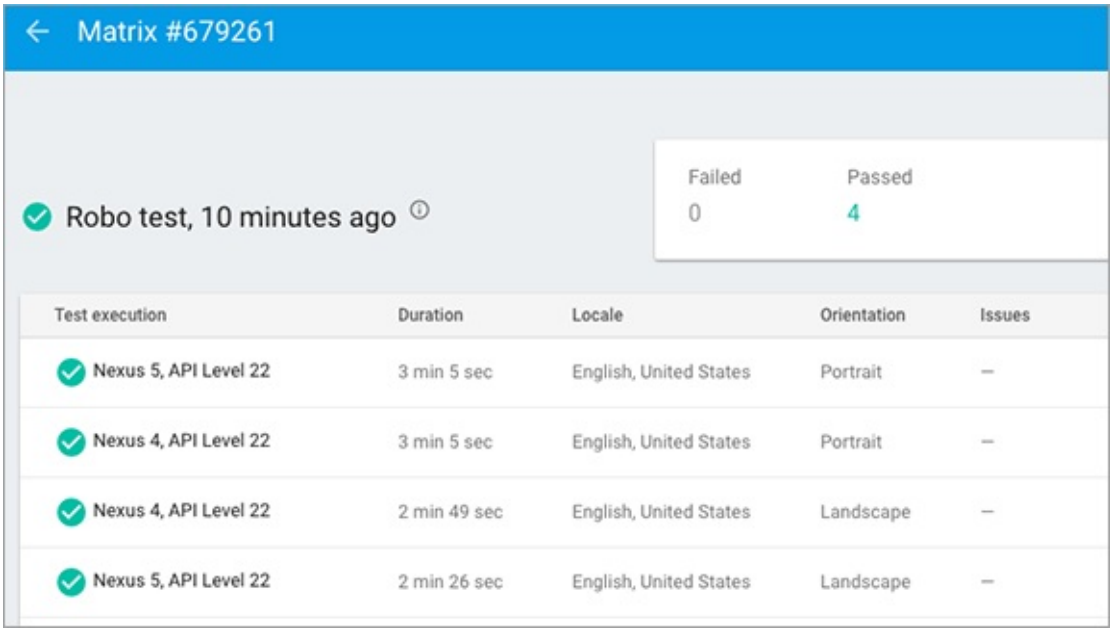
The Test Lab creates tests called **robo tests** for you, and you can also create and run your own tests. To run your app in the Test Lab:

1. In Android Studio: **Build** > **Build APK.**
2. Locate the APK in your computer's file system. It will likely be in `app/build/apk` or `app/build/outputs/apk` in the directory where the Android Studio project is saved.

   On Mac, you can click **Reveal in Finder** to see the location of the APK.
3. In the Firebase console select **Test Lab** > **Run a Test**.
4. Upload your APK.
5. From the matrix of devices and API levels that appears, choose the combination of devices to run your app on.



6. Choose **Start N Tests** at the bottom right of the device matrix, where N is the valid combination of devices and API levels you have chosen.
7. View the report when the tests have finished.



As well as running the auto-generated "robo" tests, you can also run your own instrumentation tests, which are tests you have written specifically to test your app. For example, you can run Espresso tests in the Test Lab.

When you write instrumentation tests, you create a second APK to upload to Test Lab along with the APK for your app.

Learn more at Firebase Test Lab for Android Overview.

Firebase Demo

There is a public Firebase demo project that you can use to explore the Firebase console. The Firebase demo project is a standard Firebase project with fully functioning analytics, crash reporting, test lab, and more. Anyone with a Google account can access it. It's a great way to look at real app data and explore the Firebase feature set.

You cannot send notifications from the demo project, because you do not have owner access to it. Nor can you see and modify data in the database.

The data in the demo project is real data from an app called Flood-It. You can download this app and use it to contribute to the data yourself. Flood-It is a simple game, where you see how quickly you can cover the board with a single color. Flood-It was created by Lab Pixies, which is now a Google company.

Go ahead and download and play Flood-It if you want, but don't forget to come back and keep learning! Get the Flood-It app from Google Play here.

Learn how to access the Firebase demo in the Firebase help center.

More Firebase features

Firebase has many more tools to help you develop your app and help it reach your audiences. However, our goal in discussing Firebase in this course is to introduce you to Firebase and make you aware of its features, and to inspire you to learn more about Firebase on your own.

To get started learning on your own about Firebase:

- Do the "Firebase in a Weekend" online course www.udacity.com/course/ud0352
- Work through the Firebase codelab codelabs.developers.google.com/codelabs/firebase-android

Learn more about Firebase

Get started with Firebase

- Firebase console
- Add Firebase to your Android project
- Firebase public demo
- Firebase in a Weekend online course
- Firebase codelab

Firebase Analytics

- Add Firebase Analytics to your Android app
- Key metrics available in the Firebase Analytics dashboard
- Predefined events: FirebaseAnalytics.Event
- Predefined parameters: FirebaseAnalytics.Param

Firebase Notifications

- Firebase notifications

Firebase Database

- Firebase database
- Set up Firebase Realtime Database for Android
- Read and Write Data on Android

Firebase Test Lab

- Firebase Test Lab for Android
- Use Firebase Test Lab for Android from the Firebase Console

Make money from your app

It's exciting to build an app and see it run. But how do you make money from your app?

First, you need to create an app that works well, is fast enough, doesn't crash, and is useful or entertaining. It must be compelling so users will not only want to install and use it, but will want to keep on using it.

Assuming your app is robust and provides features that are useful, entertaining, or interesting, there are various ways for you to make money from your app.

Ways to make money

The monetization models are:

- Premium model—users pay to download app.
- Freemium model:
  - downloading the app is free.
  - users pay for upgrades or in-app purchases.
- Subscriptions—users pay a recurring fee for the app.
- Ads—the app is free but it displays ads.

Premium apps

Users pay up front to download premium apps. For an app that provides desirable functionality to a small, highly targeted audience, attaching a price to your app can provide a source of revenue. Be aware that some users will refuse to download an app if they have to pay for it, or if they cannot try it first free of charge. If users can find other similar apps that are available free or at a lower cost, they might prefer to download and try those other apps.

Freemium apps

A "freemium" app is a compromise between a completely free app and one that charges a fee to install. The app is available for installation free of charge either with limited functionality or for a limited duration. Your goal for a freemium app should be to convince users of the value of your app, so that after they have used it for a while, they are willing to pay to keep using it or to upgrade for more features. Have you ever downloaded a free app and then paid for the upgraded functionality? What was it you liked about the app that made you want to pay for it?

Another way to make money from a freemium app is to provide in-app purchases. For a game, your app might offer new content levels in the game, or new items to make the game more fun. Think about the mobile games you have played. Which ones have offered in-app purchases and what for? Have you ever made in-app purchases on a mobile app?

Subscriptions

With the subscription model, users pay a recurring fee to use the app. This is very similar to the premium model, except that users pay at regular billing cycles rather than once at installation time. You can set up subscriptions so that users pay either monthly or annually. If you provide your app on a subscription basis, consider offering regular content updates or some other service that warrants a recurring fee.

If you decide to offer your app on the subscription model, it's a good idea to let users have a free trial. Many users will refuse to install an app that makes them pay before they have even tried it to see if it suits their needs.

Ads

One common monetization strategy is to provide your app free of charge but run ads in it. The user can use your app as much as they like, but the app will show them ads occasionally.

If your app displays ads, always be considerate of the user. If your app shows so many ads that it annoys the user, they might stop using it or uninstall it.

Adding ads to your app is straightforward. The best way to incorporate ads into your app is to use AdMob.

AdMob

Google provides tools for advertisers to create ads and define the targeting criteria for their ads. For an example of targeting criteria, an ad might be targeted to be shown to people in a specific location. Google has a huge inventory of ads to display on both websites and mobile apps. You can display ads from this inventory in your app by using AdMob (which stands for Ads on Mobile).

To display an ad in your app, add an `AdView` in the layout of an activity and write a small amount of boilerplate code to load the ad. When a user runs your app and goes to that activity, an ad appears in the `AdView`. You do not need to worry about finding an ad to display because Google handles that for you.

How ads help you make money

Google pays you when users click ads in your app. The exact amount you get paid depends on the ads that get shown, and how much the advertisers were willing to pay for their ads. The total amount paid by the advertisers is distributed between Google and the publisher of the website or app where the ad appears.

Don't click on ads in your own app

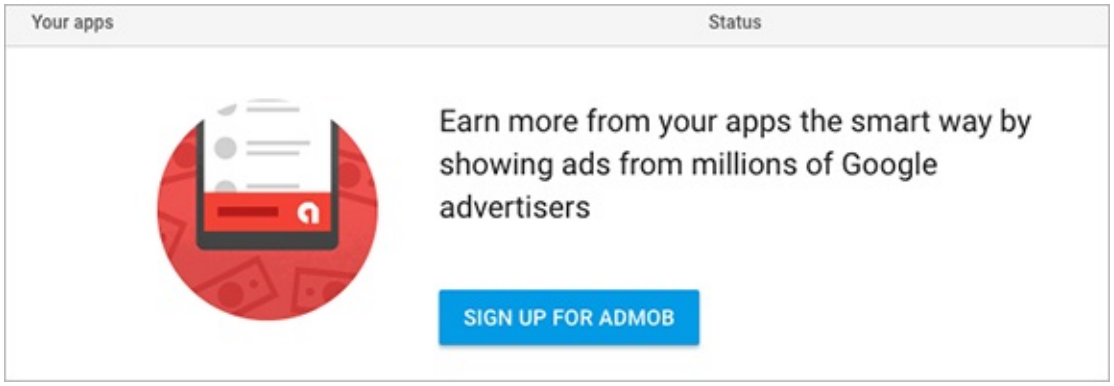
Google has policies that prevent website publishers and app publishers from clicking ads in their own websites and apps. The advertisers pay when people click their ads, so it would not be fair for you to display an ad in your app, then click the ad, and cause the advertiser to pay you for clicking the ad in your own app.

Read more about AdMob policies in the AdMob help center.

Create an AdMob account

Before you can experiment with running ads in your app, you need to enable AdMob for that app. To enable AdMob use these steps:

1. In the Firebase console, select **AdMob** in the left hand navigation, then select **Sign Up For AdMob**.


Your apps
Status
Earn more from your apps the smart way by showing ads from millions of Google advertisers
SIGN UP FOR ADMOB

You will be re-directed to the AdMob console.

1. Follow the sign up wizard to create your AdMob account and add your app to AdMob.

To display an ad in your app, you need your AdMob app ID and an ad unit ID. You can get both of these in the AdMob console.

Implement AdMob in your app

To display an ad in your app:

1. Make sure you have added your app to your Firebase project.
2. Add the dependency for firebase-ads to your app-level build.gradle (Module: app) file: (where x is the latest version) `compile 'com.google.firebase:firebase-ads:x.x.x'`
3. Add an `AdView` to the layout for the activity that will display the ad.
4. Initialize AdMob ads at app launch, by calling `MobileAds.initialize()` in the `onCreate()` method of your main activity.
5. Update the `onCreate()` of that activity to load the ad into the `AdView`.

Get ready to run test ads

While you are developing and testing your app, you can display and test ads to make sure your app is setup correctly to display ads. When testing ads you need:

- Your device ID (IMEI) for running test ads. To get the device ID either:
  - Go to **Settings** > **About phone** > **status> IMEI.**
  - Dial ***#06#**.
  - Your AdMob app ID. Get this in the AdMob console.
  - An ad unit ID. Get this in the AdMob console.

Add the AdView to display the ad

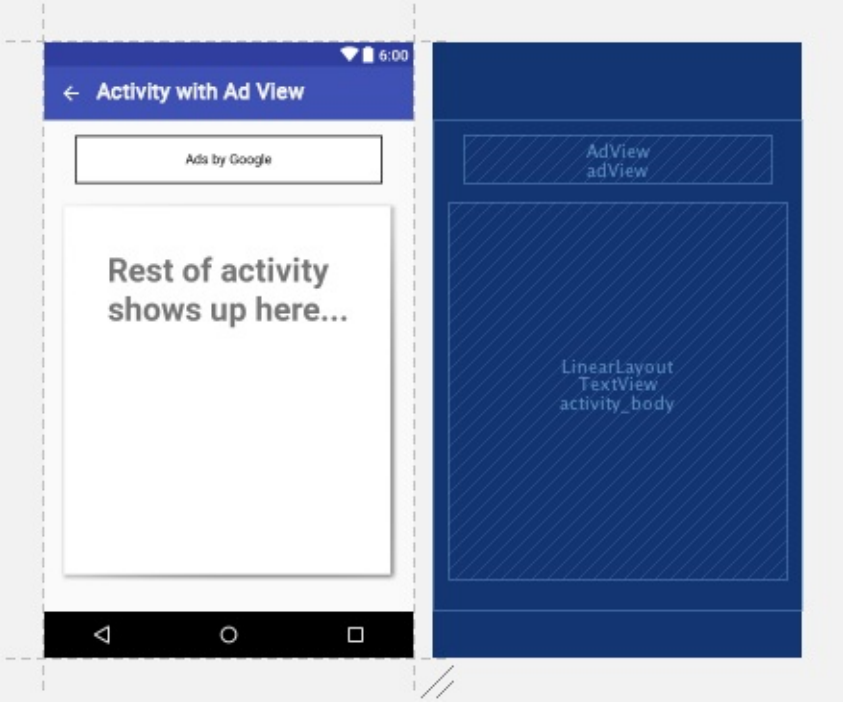
In the activity's layout file where you want the ad to appear, add an `AdView`:

```
<com.google.android.gms.ads.AdView
    android:id="@+id/adView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="16dp"
    android:layout_centerHorizontal="true"
    ads:adSize="BANNER"
    ads:adUnitId="@string/banner_ad_unit_id">
</com.google.android.gms.ads.AdView>
```

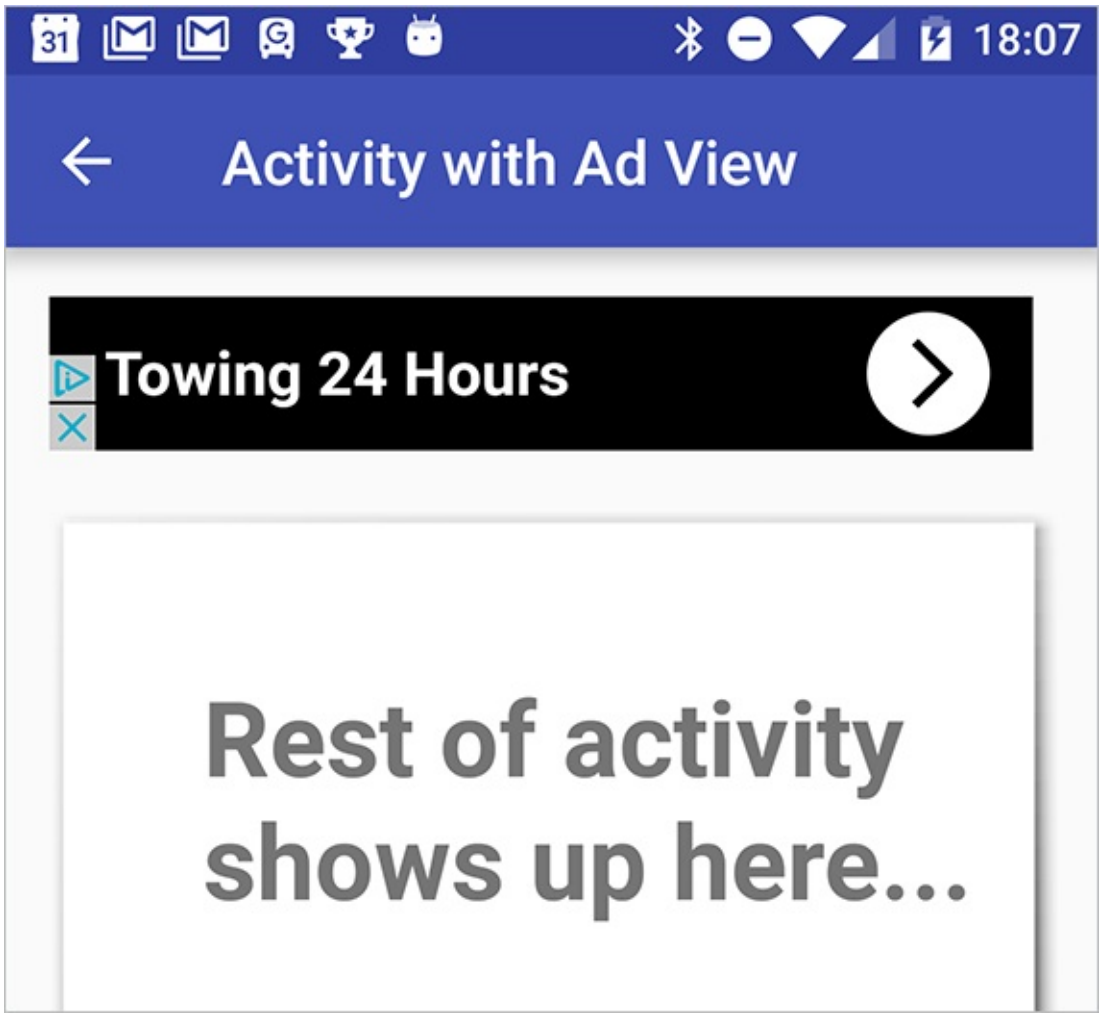
You also need to add the `ads` namespace to the root view of the layout:

```
<RootLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:ads="http://schemas.android.com/apk/res-auto"
    …>
```

Here's an example of a layout with an `AdView` in the Layout Editor:



When the app runs, the AdView is replaced by an ad.



Initialize MobileAds

The `MobileAds` class provides methods for initializing AdMob ads in your app. Call `MobileAds.initialize()` in the `onCreate()` method of your main activity, passing the context and your app ID (which you get from the AdMob console).

```
// Initialize AdMob
MobileAds.initialize(this, "ca-app-pub-1234");
```

Write the code to load the ad

In the `onCreate()` method of the activity that displays the ad, write the code to load the ad.

While you are developing and testing your app you can display ads in test mode by specifying specific test devices. To show ads on your own device, you need to device ID (IMEI), which you can get from **Settings > About phone > Status> IMEI** or by dialling ***#06#**.

To load an ad:

1. Get the `AdView` where the ad will appear.
2. Create an `AdRequest` to request the ad.
3. Call `loadAd()` on the `AdView` to load the ad into the `AdView`.

Here's the code to write in `onCreate()` in the activity:

```
// Get the AdView
AdView mAdView = (AdView) findViewById(R.id.adView);
// Create an AdRequest
AdRequest adRequest = new AdRequest.Builder()
  // allow emulators to show ads
  .addTestDevice(AdRequest.DEVICE_ID_EMULATOR)
  // allow your device to show ads
  .addTestDevice("1234") // your device id
  .build();
// Load the ad into the AdView
mAdView.loadAd(adRequest);
```

When you are ready to run ads for real, you'll need to make some changes to how you create the AdRequest object. See Get Started with Android AdMob for more information on what you need to do.

Learn more about AdMob

- Earn from your Android apps
  - Earn revenue from AdMob ads
- Get started with AdMob in Android Studio
- Set up Android for AdMob
- Sign up for AdMob
- Access AdMob console after you have signed up
- You can also access the AdMob console from the AdMob link in the Firebase console)
- AdMob help center
- AdMob policies
  - Anti-spam (don't click on ads in your own apps) policies
- AdMob glossary

15.1: Publish!

Contents:

- Prepare your app for release
- What is an APK?
- Test your app thoroughly
- Make sure your app has the right filters
- Add a launcher icon to your app
- Add an Application ID
- Specify API Level targets and version number
- Reduce your app's size
- Clean up your project folders
- Disable logging and debugging
- Reduce the size of your app's image
- Generate signed APK for release
- Publish your app!
- Create an account in the Google Play Developer Console
- Run pre-launch reports
- Review criteria for publishing
- Submit your app for publishing
- Final summary
- Learn more

In previous practicals you learned how to build and test your app, and now it's time to learn how publish it. So, what do you have to do to publish your app? This chapter covers the the high-level steps for publishing your Android app to the Google Play store, and it introduces the Google Play developer console, where you can upload your app to Google Play. It does ***not*** teach you everything there is to know about the Google Play developer console. We hope, though, that after you read this chapter you will be excited to upload your app and investigate all the different features of the console.

This chapter has two main sections:

- **Prepare your app for release** discusses the tasks you need to do to make sure your app is really ready to publish.
- **Publish!** discusses alpha and beta testing, and how to use the Google Play developer console to publish your app.

Prepare your app for release

The main goal when preparing your app for release is to make sure it is ***really*** ready. Android users expect high-quality apps that look great and work well. There will always be more features that you want to add to your app, or more features that your users ask for. There's a big difference, however, between a great app that could be made even better versus an app that breaks often, provides an incomplete experience, has navigation paths that go nowhere, or does not have a way to get to the important parts of the app.

Think about your own favorite apps. What do you like about them?

Think about apps that you have uninstalled or hardly ever use. What didn't you like about them? Think about apps that you were excited to get but then were disappointed when you started using them. What caused that disappointment for you?

The high level tasks for publishing your app to the Google Play store are:

1. Prepare the app for release.
2. Generate the signed APK.
3. Upload the APK to the Google Play developer console.
4. Run alpha and beta tests.

5. Publish to the world!

This chapter takes a high-level look at each of these tasks.

What is an APK?

An APK is a zip file that contains everything your app needs to run on a user's device. It always has the **.apk** extension. You need an APK to publish your app in the Google Play store.

You can use Android Studio to create the APK for your app. Before you generate the APK for your app, you need to do everything you can to make your app successful, including:

- Test your app thoroughly.
- Make sure your app has the correct filters.
- Add an icon.
- Choose an Application ID.
- Specify API levels targets.
- Clean up your app.

When your app is completely ready, then you can generate a signed APK to upload to the Google Play store.

Take a look at the launch checklist in the Android developer documentation.

Test your app thoroughly

As you develop your app, test it on your own Android device and in Android Studio's emulator. Make sure you test the app for different screen sizes and orientations. Also check that the app works properly on older devices.

Share your app with your developer friends. Make a zip file and send it to other developers. Then they can load your app into Android Studio and run it.

You know how your app is supposed to work, after all; you designed and built it. You know the "right way" to use your app. However, users have a really creative way of trying to use apps that you never would have believed, let alone considered. They might try to use your app or its features in ways you had not thought of, or might test it in ways you had not tested it. Encourage your testers to try out your app in different ways, to try to achieve different goals, and to take different paths navigating through the activities. This will help catch errors, inconsistencies, or functionality failures that you were not able to find, because you are too familiar with how the app is supposed to work.

Make sure you run formal tests on your app, including unit and Espresso tests. These tests should cover both the core features of your app, and the main integration points where your app calls out to another API or retrieves data from the web. These are points that are critical to your app, and they are the areas of code likely to break.

Use Firebase Test Lab to run your app on a range of real devices in Google's data centers. This way you can verify both the functionality and the compatibility of your app across many different kinds and versions of devices before to releasing your app to a broader audience.

Read about Firebase Test Lab for Android at firebase.google.com/docs/test-lab/.

Make sure your app has the right filters

When a user searches or browses for apps in the Google Play store, the results include only apps that are compatible with the user's device. For example, if a person uses a phone that has a small screen, Google Play's search results do not include apps that require a large TV-sized screen.

Make sure your app specifies the appropriate requirements to ensure that it reaches the right audience. For example, if your app requires biometric hardware for reading fingerprints, then add the requirement in the Android manifest.

```
<uses-feature android:name="android.hardware.fingerprint"/>
```

However, specifying that your app needs a fingerprint reader limits the audience for your app to people who have devices with a fingerprint reader. You should think carefully before adding restrictions to the manifest that might limit who can see and download your app.

If your app really does need a particular attribute to be present on the user's device, then make sure to include that restriction in the manifest, to ensure that everyone who can find and download your app can actually run it. People are very likely to give your app a bad review if they install it only to find that it does not run on their device.

Hardware filters

You can specify that your app uses hardware features, such as:

- light sensor

  ```
  <uses-feature android:name="android.hardware.sensor.light" />
  ```

- gamepad

  ```
  <uses-feature android:name="android.hardware.gamepad" />
  ```

- step counter

  ```
  <uses-feature android:name="android.hardware.sensor.stepcounter" />
  ```

- and many more

See the full list at developer.android.com/guide/topics/manifest/uses-feature-element.html.

Software filters

You can specify that your app requires the device to have software features such as:

- a particular shared library is installed. For example:

  ```
  <uses-library android:name="com.google.android.maps"/>
  ```

- uses a minimum Android API level. For example:

  ```
  <uses-sdk android:minSdkVersion="19">
  ```

Countries

In the process of uploading your app to Google Play, you can select the countries where your app will be available. If you specify this, then only users in those countries will be able to find and download your app.

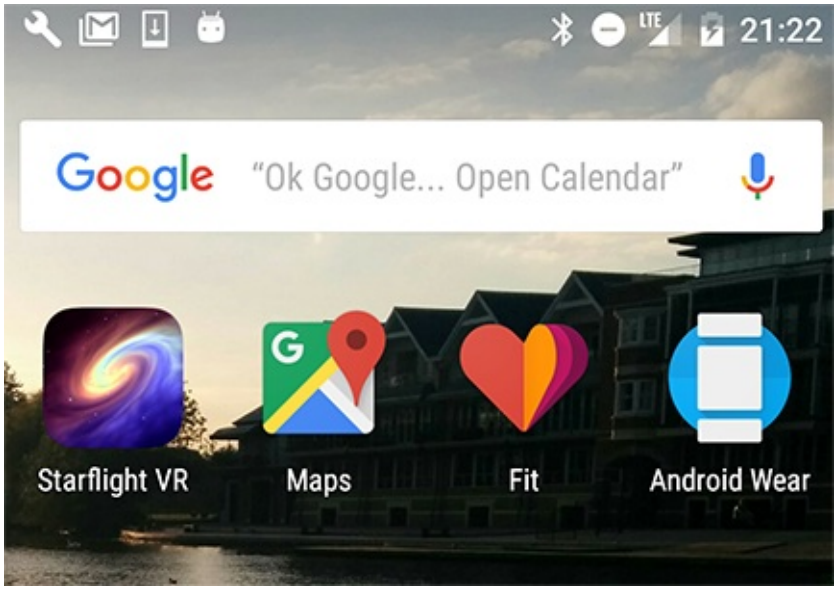
Add a launcher icon to your app

A launcher icon is a graphic that represents your application. The launcher icon for your app appears in the Google Play store listing. When users search the Google Play store, the icon for your app appears in the search results.

When a user has installed the app, the launcher icon appears on the device in various places including:

- On the home screen
- In **Manage Applications**

- In **My Downloads**



Read the Launcher Icons design guide for advice on designing your launcher app to encourage users to use your app.

Add an Application ID

The Application ID uniquely identifies an application. Make sure your app has an Application ID that will always be unique from all other applications that a user might install on their device.

When you create a project for an Android application, Android Studio automatically gives your project an Application ID. The value is initially the same as the package for the app. The Application ID is defined in the build.gradle file. For example:

```
defaultConfig {
   applicationId "com.example.android.materialme"
   minSdkVersion 15
   targetSdkVersion 24
   versionCode 1
   versionName "1.0"
}
```

You can change your app's Application ID. It does not have to be the same as your app's package name. At times as you worked through the practicals in this course, you created a copy of an Android Studio project. After copying the project, you changed the Application ID to make sure that it was unique when you installed the app on your device.

When you are getting ready to publish your app, review the Application ID. The Application ID defines your application's identity. If you change it, then the app becomes a different application and users of the previous app will not be able to update to the new app.

Specify API Level targets and version number

When you create a project for an Android app in Android Studio, you select the minimum and target API levels for your app.

- **minSdkVersion** — minimum version of the Android platform on which the app will run.
- **targetSdkVersion** — API level on which the app is designed to run.

You can set these values in the Android manifest file, and also in the app-level build.gradle file.

Note: The value in build.gradle **overrides** the value in the manifest file. To prevent confusion, we recommend you put the values in build.gradle, and remove them from the manifest file,. Setting these attributes in build.gradle also allows you to specify different values for different versions of your app.

When you get your app ready for release, review API level targets and version number values and make sure they are correct. People will not be able to find your app in the Google Play store if they are using devices whose SdkVersion is below the value specified in your app.

Here's an example of setting these attribute values in build.gradle:

```
android {
  ...
  defaultConfig {
    ...
    minSdkVersion 14
    targetSdkVersion 24
}
```

Note: The values of minSdkVersion and targetSdkVersion are the level of the API, not the version number of the Android OS.

Codename		Version	Release Date	API Level
Honeycomb		3.0 - 3.2.6	Feb 2011	11 - 13
Ice Cream Sandwich		4.0 - 4.0.4	Oct 2011	14 - 15

Jelly Bean		4.1 - 4.3.1	July 2012	16 - 18
KitKat		4.4 - 4.4.4	Oct 2013	19 - 20
Lollipop		5.0 - 5.1.1	Nov 2014	21 - 22
Marshmallow		6.0 - 6.0.1	Oct 2015	23

Nougat		7.0	Sept 2016	24

Version number

You need to specify a version number for your app. As you improve your app to add new features, you will need to update the version number each time you release a new version to the Google Play store. Read more in the Android Version guide.

Product Flavors

You can generate different "product flavors" for your app. A product flavor is a customized version of the application build. For example, you could have a demo version and a production version. Here's an example of how to define product flavors in build.gradle:

```
android {
  ...
  productFlavors {
    demo {
      applicationId "com.example.myapp.demo"
      versionName "1.0-demo"
    }
    full {
      applicationId "com.example.myapp.full"
      versionName "1.0-full"
    }
    }
}
```

Read more about product flavors in the Build Configuration developer guide.

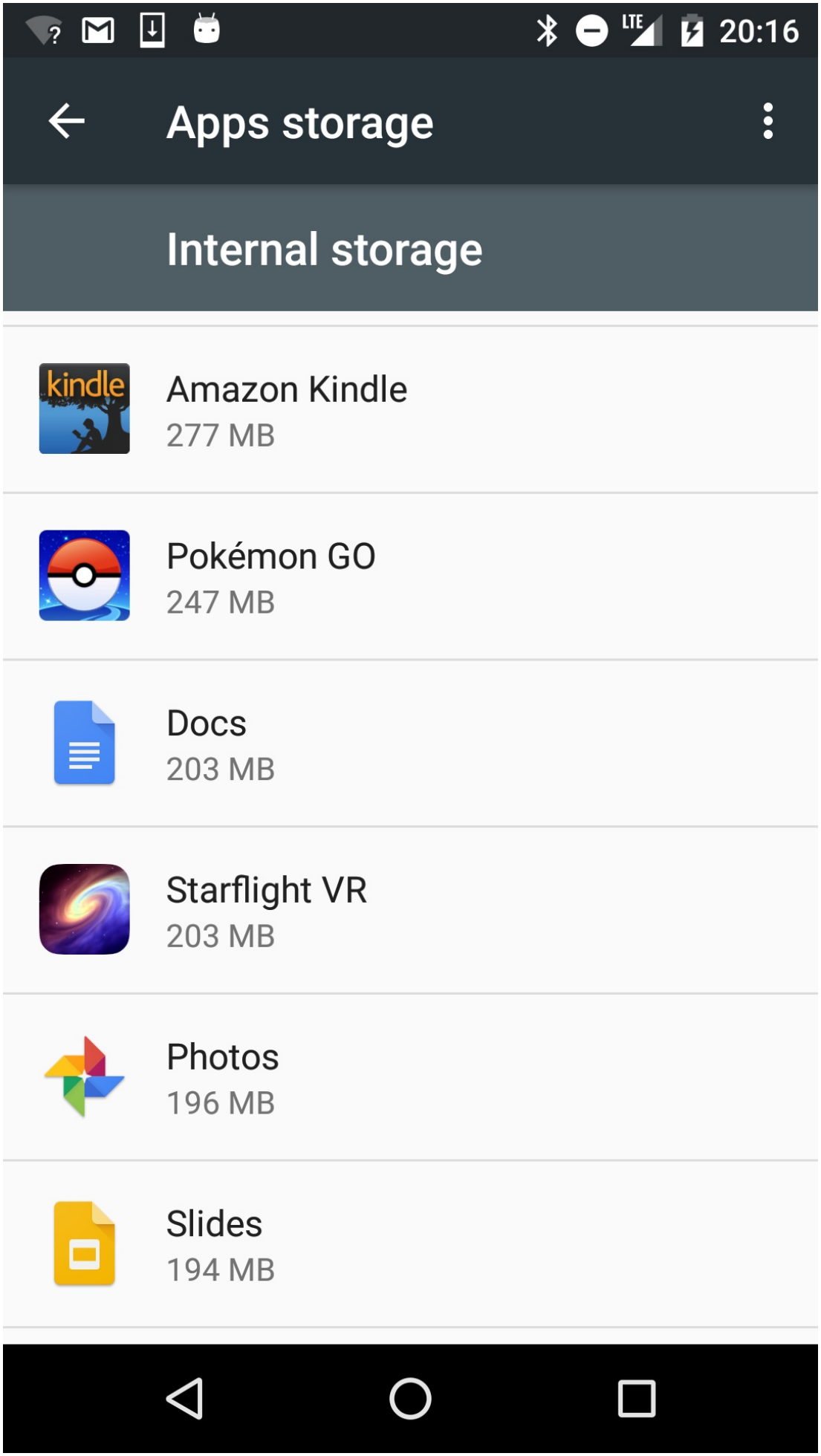
Reduce your app's size

The size of your APK affects:

- how fast your app loads
- how much memory it uses
- how much power it consumes

The bigger the size of your app's APK, the more likely it is that some users will not download it because of size limitations on their device or connectivity limitations. Users who have pay-by-the-byte plans will be particularly concerned about how long an app takes to download. If your app takes up too much space, users will be more likely to uninstall it when they need space for other apps or files.

Take a look at the apps on your own Android phone now. Which apps take up the most space? If you ran out of space on your Android phone, which of the apps that you downloaded would you uninstall?


20:16
Apps storage
Internal storage
Amazon Kindle
277 MB
Pokémon GO
247 MB
Docs
203 MB
Starflight VR
203 MB
Photos
196 MB
Slides
194 MB

There's no magic to minimizing the size of your app's APK, most of what you need to do is common sense. For example:

- Clean up your project to remove unused resources
- Re-use resources
- Minimize resource use from libraries
- Reduce native and Java code
- Reduce space needs for images

Clean up your project folders

The first step in making your app as small as possible is to clean up your project folders so that when you create the APK, Android Studio includes only the files that your app needs.

An APK file consists of a ZIP archive that contains all the files that comprise your app. These files include Java class files, resource files, and a file containing compiled resources.

The high level project folders in your project are:

- **src**: This folder contains the source files for your app. Remove any Java files that are not used. Make sure the folder does not contain any .jar files.
- **lib**: This folder contains third-party or private library files, including prebuilt shared and static libraries (such as .so files). Make sure you remove any unused library files.
- **jni**: This folder contains native source files associated with the Android Native Developer Kit, such as .c, .cpp, .h, and .mk files.
- **res**: The res sub folders contain the resources such as layouts, colors, strings, styles that your app uses.

Note: While you are developing your app, it can be easy to create additional resources that your app ends up not using, so be sure to check for and remove unused resources.

Disable logging and debugging and check production URLs

While you are developing your app, you are no doubt testing it carefully, perhaps adding unit tests, showing toasts, and writing logging statements. When you prepare your app for release, you need to remove all the extra code.

To disable logging and debugging:

- Remove logging statements and calls to show toasts
- Remove all Debug tracing calls from your source code files such as startMethodTracing() and stopMethodTracing().
- Disable debugging in the Android manifest by either:
- Removing android:debuggable attribute from tag
- Or setting android:debuggable attribute to false
- Remove unused tests

Also, make sure that if your application accesses remote servers or services, it uses the production URL or path for the server or service and not a test URL or path. Likewise, many companies with public APIs have test permissions and production level permissions. Make sure that your security or passwords for accessing these servers are production level as well.

Reduce the size of your app's images

Reducing the size of the images in your app can go a long way to reducing the size of the APK file.

- First, make sure your app does not contain any image resources it does not use.
- For each static image your app uses, you need to create separate versions of that image for all screen sizes that your app might run on. However, in some cases, you can use Drawable objects, VectorDrawables, and 9-patch files instead.

- If your app uses static images, be sure to crunch PNG files, and compress both PNG and JPEG files to minimize their size. You can use Google to search for crunching, crushing, and compressing tools for images.
- Consider using WebP format for images. Android supports WebP from Android 4.0+.

Read more about how to reduce the size of your app in the Reduce APK size guide.

Note: Drawable objects are defined in XML. Android draws them when the app needs to display them, which means your app does not need to store images for them. However, because Android generates them as needed, it can take longer for the images to appear on screen, so it's best to use Drawable objects for smaller images such as icons and logos. Drawable objects do not support the same complexity and detail that you can get with bitmaps.
In Android 5.0 (API Level 21) and above, you can define *vector drawables*, which are images that are defined by a path. Vector drawables scale without losing definition. Most vector drawables use SVG files, which are plain text files, or compressed binary files that include two-dimensional coordinates for how the image is drawn on the screen. Because SVG files are text, they use less space than most other image files. Also, you only need one file for a vector image instead of a file for each screen density.

Consider the use case for your images, and use Drawable objects and 9-patch files wherever it makes sense. See the *Drawable, Styles and Themes* lesson in this course for more information.

Generate the signed APK for release

When your app is ready to upload to Google Play, you must generate and sign the APK for your app. Android Studio has tools for generating your APK and signing it with a digital certificate.

When Android Studio signs the app, it creates a public certificate and a private key. It attaches the public certificate to the APK. You must securely store the private key in a keystore

The public-key certificate serves as as a "fingerprint" that uniquely associates the APK to you and your corresponding private key. This helps Android ensure that any future updates to your APK are authentic and come from you, the original author.

For information about digital certificates, storing your private key, and generating the digitally-signed APK, see the guide Sign Your App.

Publish your app!

When you've tested your app, cleaned it up, reduced its size, and generated the APK, you are ready to publish it to Google Play.

After you upload your app to Google Play, you can run alpha and beta tests before releasing it to the public. Running alpha and beta tests lets you share your app with real users, and get feedback from them. This feedback does not appear as reviews in Google Play.

Run alpha tests while you are developing your app. Use alpha tests for early experimental versions of your app that might contain incomplete or unstable functionality. Running alpha tests is also a good way to share your app with friends and family.

Run beta tests with limited number of real users, to do final testing before your app goes public.

Once your app is public, users can give reviews. So, make sure you test it thoroughly before putting it out on Google Play for anyone to download.

For more information on alpha and beta tests see:

- Developer guide: developer.android.com/distribute/engage/beta.html
- Help center: support.google.com/googleplay/android-developer/answer/3131213

Create an account in the Google Play Developer Console

Whether you want to run alpha and beta tests, or publish your app to the public on Google Play, you need to upload your APK in the Google Play developer console.

Go to the console at **play.google.com/apps/publish/.**

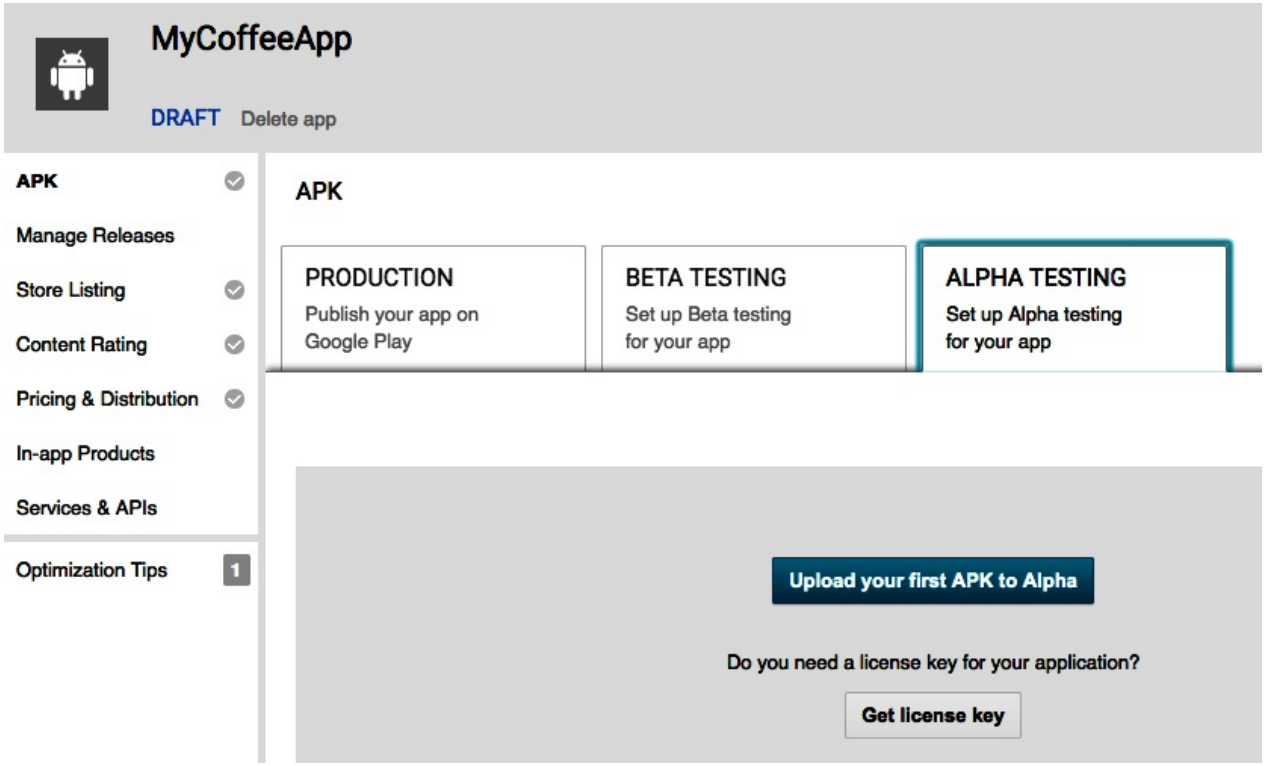
To begin, get a Google Play developer account. You will need to pay for the account. The high-level steps are:

1. Go to play.google.com/apps/publish/
2. Accept the agreement.
3. Pay the registration fee.
4. Enter your details, such as your name, address, website, phone and email preferences.

When you have set up your account, you can upload your APK. In the Google Play Developer console interface, choose:

- Production
- Beta Testing
- Alpha Testing

Then you can browse for the APK to upload, or drag and drop it to the console.



You need to satisfy the following requirements before you can publish your app to the public:

- add a high-res icon
- add a feature graphic (in case your app is selected as a Featured App in Google Play)
- add 2 non-Android TV screenshots
- select a category
- select a content rating
- target at least one country
- enter a privacy policy URL
- make your app free or set a price for it
- declare if your app contains ads
- add a required content rating

This list might seem long, but the Google Play developer console helps you figure out if your app is ready to launch. Click the "Why can't I publish?" link to find out what else you need to do to publish your app.

Run pre-launch reports

After you upload your APK, you can run pre-launch reports to identify crashes, display issues and security issues. During the pre-launch check, test devices automatically launch and crawl your app for several minutes.

The crawl performs basic actions every few seconds on your app, such as typing, tapping, and swiping. The pre-launch tests use Firebase Cloud Test Lab.

For more information on pre-launch supports, see the Google Play Help center article Use pre-launch reports to identify issues.

Review criteria for publishing

Your app must comply with Google Play policies, which ensure that all apps on Google Play provide a safe experience for everyone.

There are policies governing

- Restricted content
- Intellectual property, deception and spam
- Privacy and security
- Monetization and ads
- Store listing and promotion
- Families

Learn more at play.google.com/about/developer-content-policy/

Restricted content

Google Play does not allow apps that are sexually-explicit, hateful, racist, encourage bullying or violence, or facilitate gambling.

https://play.google.com/about/restricted-content/

Intellectual property, deception and spam

Google Play does not allow apps that are not honest. In other words, they pretend to be other apps or pretend to come from other companies or impersonate other brands. Google Play does not allow apps that attempt to deceive users. Google Play does not allow apps that spam users such as apps that send users unsolicited messages. Google Play does not allow apps that are duplicative or of low-quality.

https://play.google.com/about/ip-deception-spam/

Privacy and security

Google Play requires that your app treats users' data safely and keeps private user information secret. If your app accesses or transmits private data, it must publish a statement about how it uses users' data.

Google Play does not allow apps that damage or subversively access the user's device, other apps, servers, networks, or anything that it should not. Basically, your app should not interfere with anything else, or cause any damage to anything, or try to access anything that it does not have authorization to access.

Google Play does not allow apps that steal data, secretly monitor or harm users, or are otherwise malicious.

https://play.google.com/about/privacy-security/

Monetization and Ads

Google Play has rules regarding accepting payment for in-store and in-app purchases.

Google Play does not allow apps that contain deceptive or disruptive ads.

https://play.google.com/about/monetization-ads/

Store Listing and Promotion

Publishers must not attempt to promote their own apps unfairly. For example, you are not allowed to get 100,000 of your closest friends to give your app a 5 star rating so that it appears with very favorable reviews. Your app's icon, title, description and screenshot must all fairly represent your app, and not make any exaggerated or misleading claims.

In other words, don't cheat to get a better Google Play rating or placement.

https://play.google.com/about/storelisting-promotional/

Submit your app for publishing

When you upload your app for production, Google checks your app. Google runs both automatic and manual checking on your app.

If your app is rejected, fix the problem and try again!

Final summary

You have reached the end of this course. We hope you have enjoyed the journey and that you feel ready to go build your own Android apps. We look forward to seeing your apps in the Google Play store!

Learn more

Preparing your app

- Preparing for release developer.android.com/studio/publish/preparing.html
- Launch checklist developer.android.com/distribute/tools/launch-checklist.html
- Core app quality checklist developer.android.com/distribute/essentials/quality/core.html
- Handling user data play.google.com/about/privacy-security/user-data/
- App filters developer.android.com/google/play/filters.html
- Min, max, and target API levels developer.android.com/guide/topics/manifest/uses-sdk-element.html
- Product flavors developer.android.com/studio/build/index.html
- Version your app developer.android.com/studio/publish/versioning.html
- Google Play filters developer.android.com/google/play/filters.html
- Reduce app size developer.android.com/topic/performance/reduce-apk-size.htm
- Sign your app developer.android.com/studio/publish/app-signing.html

Google Play Developer Console

- Go to the console: play.google.com/apps/publish/
- Dev guide: developer.android.com/distribute/googleplay/developer-console.html
- Help center: support.google.com/googleplay/android-developer/#topic=3450769
- Get started publishing developer.android.com/distribute/googleplay/start.html

- Firebase Test Lab: firebase.google.com/docs/test-lab/

Alpha and Beta testing

- Dev guide: developer.android.com/distribute/engage/beta.html
- Help center: support.google.com/googleplay/android-developer/answer/3131213